Journal of Systems Architecture xxx (2010) xxx–xxx

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

The Policy Machine: A novel architecture and framework for access control policy specification and enforcement

David Ferraiolo [a], Vijayalakshmi Atluri [a,b,*], Serban Gavrila [a]

[a] *National Institute of Standards and Technology, 100 Bureau Dr. Stop 8930, Gaithersburg, MD 20899, USA*
[b] *MSIS Department and CIMIC, Rutgers University, USA*

ARTICLE INFO

Article history:
Received 15 October 2009
Received in revised form 18 March 2010
Accepted 22 April 2010
Available online xxxx

Keywords:
Security policy enforcement framework
Policy Machine
Access control

ABSTRACT

The ability to control access to sensitive data in accordance with policy is perhaps the most fundamental security requirement. Despite over four decades of security research, the limited ability for existing access control mechanisms to generically enforce policy persists. While researchers, practitioners and policy makers have specified a large variety of access control policies to address real-world security issues, only a relatively small subset of these policies can be enforced through off-the-shelf technology, and even a smaller subset can be enforced by any one mechanism. In this paper, we propose an access control framework, referred to as the Policy Machine (PM) that fundamentally changes the way policy is expressed and enforced. Employing PM helps in building high assurance enforcement mechanisms in three respects. First, only a relatively small piece of the overall access control mechanism needs to be included in the host system (e.g., an operating system or application). This significantly reduces the amount of code that needs to be trusted. Second, it is possible to enforce the precise policies of resource owners, without compromise on enforcement or resorting to less effective administrative procedures. Third, the PM is capable of generically imposing confinement constraints that can be used to prevent leakage of information to unauthorized principals within the context of a variety of policies to include the commonly implemented Discretionary Access Control and Role-Based Access Control models.

Published by Elsevier B.V.

1. Introduction

The ability to control access to sensitive data in accordance with policy is perhaps the most fundamental security requirement. While over the past four decades security researchers have proposed a large variety of policies and policy models to address real-world security problems, only a small subset of these policies are enforceable through commercially available mechanisms, and even a smaller subset can be enforced by any one mechanism. Well known examples of access control models include Discretionary Access Control (DAC) and Mandatory Access Control (MAC) [1] of the 1970s, Chinese Wall policy [2] of late 1980s, Role-Based Access Control (RBAC) [3–5] of the early 1990s, and more recently XACML-conforming services and applications [6]. It is important to recognize that within this progression of this development, a latter framework does not necessarily subsume the policy support of a former.

In this paper, we propose an access control framework, referred to as the Policy Machine (PM). The PM is not an extension of any existing model or framework, but instead is a redefinition of access control in terms of a standardized and generic set of relations and functions that are reusable in the expression and enforcement of policies. Its objective is to provide a unifying framework to support a wide range of attribute-based policies or policy combinations through a single mechanism that requires changes only in its data configuration. PM can be thought of as a logical "machine" comprising of a fixed set of data relations and functions. The relations are configurable through a fixed set of administrative operations for the expression of combinations of many policies. The functions aid in making access control decisions, and enforcing the expressed policies. Under the PM, policies are enforced through a reference mediation function. In its simplest and most general form, the PM standard architecture is comprised of one or more PM clients, one or more PM servers, a PM database, and one or more resource servers.

To enable flexible policy enforcement, PM abides by the principle of separation of policy from mechanism that dictates that no policy should be tightly coupled to its mechanism. The challenge is to identify a minimal set of primitives that can specify and enforce a large variety of attribute-based security policies. We have identified a surprisingly small set of data, relations and functions that are reusable in the expression and enforcement of a wide

* Corresponding author at: MSIS Department, Rutgers University, 1 Washington Park, Newark, NJ 07102, USA.
E-mail addresses: dferraiolo@nist.gov (D. Ferraiolo), atluri@rutgers.edu (V. Atluri), gavrila@nist.gov (S. Gavrila).

1383-7621/$ - see front matter Published by Elsevier B.V.
doi:10.1016/j.sysarc.2010.04.005

range of attribute-based policies. These include assignment relations, prohibition relations, and obligations.

Employing PM helps in building high assurance enforcement mechanisms in three respects. First, only a relatively small piece of the overall access control mechanism needs to be included in the host system (e.g., an operating system or application). This significantly reduces the amount of code that needs to be trusted.

Second, policies are enterprise- and mission-specific. The enforcement mechanisms offered by commercial products are typically limited to a few fixed policies. Organizations therefore have to resort to implementing the unsupported policies as application code or simply ignore them. Under the proposed PM framework, the desired security policy can be enforced independent of that offered by the vendors. Moreover, it is possible to protect a resource under multiple security policies simultaneously. The actions of users and processes may be controlled under multiple policies, as well as enterprise objects may be protected under multiple policies.

Third, the PM has the ability to generically impose confinement constraints to not only meet the policy objectives of Mandatory Access Control, but also to prevent the leakage of information within models that are typically incapable of imposing such constraints. This is a particularly important assurance aspect given the general threat of malware and malicious or complacent user actions. Specifically, it is well known that DAC and RBAC are vulnerable to Trojan Horse attacks. In this paper, we demonstrate how the PM framework can address such vulnerabilities.

To demonstrate PM's viability and its flexible policy support, we have developed a reference implementation. We can now show, through our reference implementation, how to configure the PM for the expression and enforcement of a diverse set of policies to include instances, and combinations of DAC, MAC, RBAC, Chinese Wall [2], ORCON [7], history-based SoD, etc.

This paper is organized as follows: In Section 2, we present the PM system architecture. In Section 3, we describe the PM's basic data sets, relations and reference mediation function. In Section 4, we show by example the PM's ability to express and enforce the control objectives of a variety of policies, including RBAC, MAC, DAC and Chinese Wall, through the configuration of PM relations. It is our hope that the reader can envision configurations of other policies as well. In Section 5 we demonstrate the PM's natural ability to combine policies. Section 6 discusses related work, and Section 7 concludes the paper by presenting a short discussion of the future vision, benefits and other functionalities of the PM.

2. System architecture

Any access control enforcement framework should comprise a policy enforcement point (PEP) and a Policy Decision Point (PDP). Normally, both these functions are performed by the OS or the application software. As such, they must be rendered as trusted components. As it is evident, since the application and OS have many other functional components, they tend to be significantly large, resulting in a large amount of code that needs to be trusted.

The PM is comprised of a PM server and PM clients, as shown in Fig. 1. The PM server includes a PM Database, a PDP, a Policy Administration Point (PAP) and an event processing module. The PM client is comprised of a PEP, its application programming interfaces (API), and PM-aware applications. In our current implementation, the PEP is implemented as a kernel simulator. Because the PEP is simple and straightforward to implement, establishing trust in the PM client is simplified. The PM server also needs to be trusted. Because it could be implemented as a central server that communicates only with PEPs and only through well defined interfaces, it has a reduced attack surface. The unshaded blocks of Fig. 1 indicate the untrusted components and the shaded blocks the trusted components.

The PM client or the user environment is the context in which the user's PM processes run. It can be an operating system, an application (e.g., a database management system), a service in a service oriented architecture, or a virtualized environment. Typically, a PM user logs on the PM by using a GUI provided by the PM client. A successful login opens a user session on the PM client. Within that session, PM presents the user with a logical view, called "personal object system" (POS), of all his/her accessible

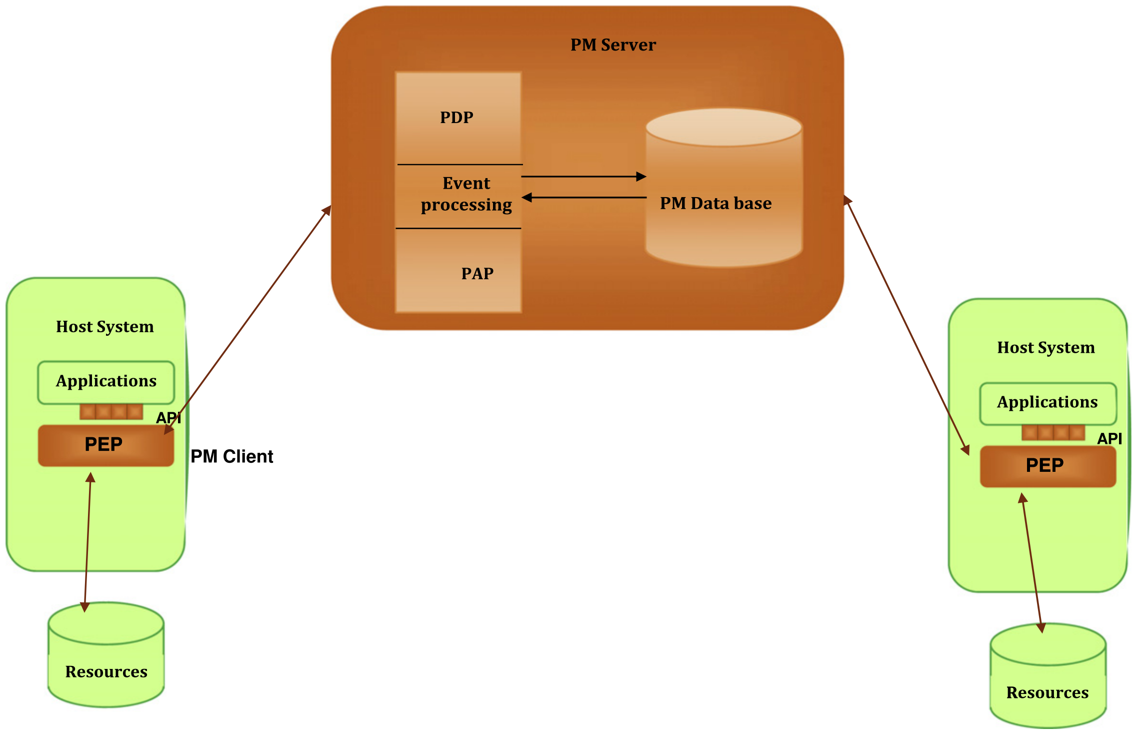

Fig. 1. The system architecture.

resources (e.g., files, e-mail messages, work items). As an alternative, PM may let the user ask for specific accessible resources. Within the session, the user may create and run various PM processes that request access to PM-protected resources. The PEP within the PM client traps each access request, and then asks the PDP within the PM server to decide whether to grant or reject the access request. The physical location of each object is known to the PM server (and transparent to the client). In the case of a granted request, the server response is accompanied by the physical location of the object. The PM client next enforces the server's decision, granting or rejecting the process' access to the object. For a granted request the PM client requires the cooperation of a resource server in performing the granted operation on the physical content of an object. The resource server can reside on the machine that includes the user environment or on a server that is dedicated the storage of PM resources. The PM client exposes a standard set of APIs that can be used to develop PM-aware applications.

The PDP receives an access request from (the PEP of) a PM client, computes a decision to grant or reject the access request, and returns the result. In this aspect, the PM server implements PM's reference mediation function. The decision is based on the identity of the user/process that issued the request, the requested operation, the requested resource/object, and the permission and prohibition relations, which are stored in the PM database. Essentially, a request is granted if and only if there exist appropriate assignment relations (see Section 3) and there does not exist a prohibition for that user or process on the requested object. The event processing module of the PM server executes the responses to events specified in the obligation relations stored in the PM database, when the events are triggered by a client's successful execution of an operation on a PM object. Finally, the administrative module (called Policy Administration Point, or PAP), is used to administer the PM database, i.e., to configure the current policy. The PM server exposes a standard set of commands that can be used by clients to solicit its services.

The PM database contains a standard set of data and relations that represent the current policy configuration. As already stated, the policy configuration is used by the PM server's PDP to compute access control decisions. The policy configuration can be set up and/or updated by a PM administrator using the PM server's PAP. The PM server's event processing module may dynamically update the policy configuration by executing the responses attached to events that were detected in the PM clients.

Finally, the PM uses the resource repositories and servers to store and retrieve the physical contents of its objects. The repository of each object is obviously known to the PM, but transparent to the user. If a client's request to access a resource is granted, this decision arrives at the client's PEP accompanied by the physical location of the object. The PM client cooperates with the resource server in transferring the contents of an object from/to a repository to/from the PM user environment where the requesting process resides. Note that the resource servers may be implemented as PM client modules.

The state of the Policy Machine may dynamically change as a consequence of successful accesses. A policy can be created through data configuration alone. However, PM has the provision to import existing policies through policy libraries.

3. The Policy Machine

3.1. PM basic elements

The PM's basic elements include authorized users (U), processes (P), system operations (Op), and objects (O). Users are unique entities that are either human beings or "system users". Objects are names that uniquely specify system entities that are controlled under one or more policies. Included in the set of objects are PM access control data and relations. The set of objects may pertain to environment specific entities such as files, ports, clipboards, e-mail messages, records and fields. The selection of entities included in this set is a matter of choice determined by the protection requirements of the system. Operations are unique actions that can be performed on the contents of objects. Some of these operations are specific to the environment for which the PM is implemented. For example, common operating system operations include read (r) and write (w). Others pertain to PM administrative operations that create and delete PM data and relations.

Human users submit access requests through processes. Most models treat users and their processes uniformly, under the concept of a subject, defined as an active entity. The PM is different in this regard by treating users and processes as independent but related entities. The impact is greater user access flexibility and transparency. A process is a system entity, with memory, and operates on behalf of a user. Essential properties of processes are that they issue access requests, have exclusive access to their own memory and none to any others, but may communicate and exchange data with other processes through a physical medium such as the system clipboard or sockets. A user may be associated with one or more processes, while a process is always associated with just one user. We denote by $process\_user(p)$ the user associated with process $p \in P$. We denote by $\langle op, o \rangle_p$ a process access request, where $op \in Op$, $o \in O$, and $p \in P$.

To afford appropriate mappings of attributes to policies, we introduce policy classes. We assume PC be the set of policy classes. A user, object or their respective attributes may belong to a policy class $pc \in PC$. Note that an object may be protected under more than one policy class, and similarly a user may belong to more than one policy class.

3.2. PM relations

PM relations are of three types – assignments, used to express and determine privileges, prohibitions that are expressed as user and process deny relations, and obligations that are defined as event-response relations. The configuration of assignments and prohibitions define the access state of users and processes, and the *access state* and the obligations together define the overall PM *policy state*.

3.2.1. Assignments

A PM privilege is a triple of the form (u, op, o) where $u \in U$, $op \in OP$, $o \in O$, and where (op, o) is said to be u's capability and (u, op) is said to be o's access entry. Its meaning is that user u can perform operation op on object o. As with other access control schemes, PM privileges are indirectly managed through higher level abstractions. The PM includes four such abstractions, namely user attributes (UA), object attributes (OA), operation sets, and policy classes (PC), with a binary assignment relation denoted by $\rightarrow$.

For each object $o \in O$, we assume the singleton set $\{o\}$ is an object attribute (an element of OA). We will use the name o to denote this singleton set $\{o\}$ (the object attribute) as well as the object o. Note that OA may contain other object attributes, which, as sets, may be equal to $\{o\}$, but have different names. For simplicity, with a slight abuse of the notation, we treat $O \subseteq OA$.

An important PM concept is that all user and object attributes have common semantics regardless of their kind. A user attribute is a many-to-many relation that defines a set of users on one side and a set of capabilities on the other. This definition is consistent with the formal definition of a role [4]. The same semantics can be used to define other kinds of attributes such as communities of interest, organizational units, clearance levels, etc. An object attribute is also a many-to-many relation that defines a set of objects on one side and a set of access entries on the other. These many-to-many relations can be derived from common PM assignment relations, defined as follows.

A user may be assigned to one or more user attributes ($u \to ua$). A user attribute may be assigned to another user attribute ($ua_1 \to ua_2$). An object attribute may be assigned to another object attribute ($oa_1 \to oa_2$) with the restriction that the second object attribute may not be an object. Another restriction is that any chain of user or object attribute assignments may not be a cycle. We use the notations $\to^*$ and $\to^+$ to denote a chain of 0 or more assignments and a chain of 1 or more assignments, respectively. By virtue of these assignments, both user and object attributes can sometimes be treated as containers. A user u is said to be "assigned to" or "in" a user attribute ua iff $u{\to}^+ua$, and an object o is said to be "assigned to" or "in" an object attribute oa, iff $o{\to}^*oa$.

User attributes can also be assigned to operation sets, $ua \to ops$, and operation sets can be assigned to object attributes, $ops \to oa$, where $ops \subseteq Op$. An assignment relation within one policy class $ua \to ops \to oa$ specifies that all users contained in ua can perform all operations in ops on all objects contained in oa.

Given the assignment relations described so far, there are at least two meanings for the user attribute assignment $ua_1{\to}^+ua_2$. The first, as described above, is that the set of users of ua_1 is contained in ua_2. The second is that the users of ua_1 have the capabilities associated with ua_2. The capabilities associated with a user attribute ua are those obtained through all $ua \to ops \to oa$ assignments. Similarly the object attribute assignment $oa_1{\to}^+oa_2$ has two meanings. The first is that the set of objects of oa_1 is contained in oa_2. The second is that the objects of oa_1 have the access entries associated with oa_2. The access entries associated with an object attribute oa are those obtained through all $ua \to ops \to oa$ assignments.

The PM allows for the combination of two or more policies (e.g., RBAC and MAC). However, not all users are controlled under all policies, nor are all user attributes and object attributes relevant to all policies. A user attribute or an object attribute may be assigned to a policy class, $ua \to pc$ or $oa \to pc, pc \in PC$. This type of assignment leads to mappings of users, user attributes, objects and object attributes to policy classes. A user u or user attribute ua "belongs to" or "is in" a policy class pc if there exists a chain of one or more assignments that start with that user or user attribute and ends with the policy class (i.e., $u{\to}^+pc$ or $ua{\to}^+pc$, $pc \in PC$). Similarly, an object o "is controlled under" or an object attribute oa "belongs to" or "is in" a policy class pc if there exists a chain of one or more assignments that start with that object or object attribute and ends with the policy class (i.e., $o{\to}^+pc$, $oa{\to}^+pc, pc \in PC$).

With the assignments in place, we are able to determine the existence of a PM privilege. A triple (u, op, o) where u is a user, op is an operation, and o is an object, is a PM privilege iff for each policy class pc_k under which o is controlled, user u has an attribute ua_k in pc_k, object o has an attribute oa_k in pc_k, and there exists an operation set ops containing op that is assigned to both ua_k and oa_k, as shown in Fig. 2. In this and other illustrations depicting assignment relations, we use dotted arrows to represent user attribute to operation set to object attribute assignment relations. We do this to avoid confusing these relations with other types of assignment relations.

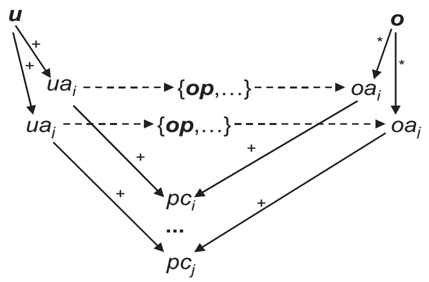

Fig. 2. (u, op, o) is a PM privilege.

3.2.2. Prohibitions

There are two types of prohibitions, user-deny and process deny. We denote by $u\_deny(u, ops, os)$ a user-based deny relation, where $u \in U, ops \in 2^{OP}$, and $os \in 2^{O}$. Its meaning is that a process executing on behalf of user u cannot perform the operations in ops on the objects in os. By specifying os as its complement, denoted by $\neg$, the meaning of $u\_deny(u, ops, \neg os)$ is that a process executing on behalf of user u can only perform the operations in ops on the objects in os. Similarly, a process-based deny relation is a triple $p\_deny(p, ops, os)$, where $p \in P$, $ops \in 2^{OP}$, and $os \in 2^{O}$. Its meaning is that the process p cannot perform operations in ops on the objects in os, and the meaning of $p\_deny(u, ops, \neg os)$ is that the process can only perform the operations in ops on objects in os. User and process denies are generally referred to as prohibitions because they represent exceptions to privileges.

3.2.3. Obligations

These obligations, also known as event pattern/response relations, define conditions and methods under which policy state data is obligated to change. An event pattern/response relation is a pair (ep, r) (usually denoted **when** ep, **do** r), where ep is an *event pattern* and r is a sequence of administrative operations, called a *response*. The event pattern specifies conditions that if matched by the context surrounding a process' successful execution of an operation on an object (an event), the administrative operations of the associated response are immediately executed, thereby changing the state of the policy. The context may pertain to and the event pattern may specify parameters like the user of the process, the operation executed, and the container(s) in which the object is included. Note that the possible formal parameters of the administrative operations comprised in the response are replaced by the appropriate values extracted from the event context. Responses are obligations performed by the PM, and as such, their execution is not predicated on privileges. Although the term obligation is used in a number of other access control frameworks such as XACML and Ponder [6,8], these frameworks either do not apply obligations in altering the state of the policy [6] and/or do not alter the access space of a process that, as we show in Section 4, is instrumental in the enforcement of a number of policies.

3.3. Administrative commands

The question remains, how are policy state data elements and relations initially created and altered in meeting policy demands. The short answer is by administrators through administrative commands. An administrative operation is specified as a parameterized procedure, whose body describes how a data set or relation (denoted by R) changes to R':

```
$opname = (x_1, \ldots, x_k)$ {
    $R' = f(R, x_1, \ldots, x_k)$
}
```

For example, consider the following administrative operation CreateUser:

```
CreateUser($u$) {
    $U' = U \cup \{u\}$
}
```

An *administrative command* is a parameterized sequence of administrative operations prefixed by a condition and has the format:

```
$cmdname(x_1, , \dots , x_k) \leftarrow$
        if (condition) then
            $aop_1$
            ...
            $aop_n$
end,
```

where $x_1, \dots , x_k (k \geqslant 0)$—are (formal) parameters and aop_1, aop_n $(n \geqslant 0)$—are administrative operations which may use $x_1, \dots , x_k$ as their parameters. The condition tests, in general, whether the user who requested the execution of the command is authorized to execute the command (i.e., the composing administrative operations), as well as the validity of the actual parameters. If the condition evaluates to false, then the command fails.

3.4. Reference mediation

The current access state is enforced by a reference mediation function. Under reference mediation, a process access request $\langle op, o\rangle_p$ is granted iff there exists a privilege (u, op, o) where $u =$ *process_user*(p), and capability (op, o) has not been denied for either u or p. It should be noted that if the PM privilege (u, op, o) exists, and if *p_deny*(p, ops, os)—exists, where $u =$ *process_user*(p), $op \in ops$, and $o \in os$, user u may be able to perform capability (op, o) through a different process p', if $u =$ *process_user*(p'). However, if *u_deny*(u, ops, os)—and privilege (u, op, o) exists where, $op \in ops$, and $o \in os$, no process of u (and therefore u) will be able to perform capability (op, o). This discussion shows how the PM resolves potential authorization conflicts: denies have precedence over privileges. Regarding potential conflicts between policies that protect an object, PM applies an "and" combination algorithm.

3.5. Consideration of inter-process communication

Although inter-process communication is not in reality a PM component, its consideration in a protection scheme is essential. In this section we describe the PM's treatment of process-to-process communication. In general, operating systems provide mechanisms for facilitating communications and data sharing between applications that provide opportunities to leak data as well. These mechanisms include but are not limited to clipboards, pipes, sockets, remote procedure calls, and messages. They all conform to a common abstraction: one process produces/creates data and inserts it into the mechanism's physical medium; the other process consumes/reads the data from the physical medium. A synchronization mechanism must also exist.

By treating the communication medium as a PM object, the PM offers strategies to support data transfer that is in compliance with the policy. For example, the producer process could create a PM object that represents the physical medium/support of the data transfer mechanism. This new object will be assigned attributes in accordance to a predefined and policy-specific set of conditions. These conditions can be specified via the event-response relations. The consumer process must be able to read the PM object that represents the physical medium under the rules of the reference mediation. A practical example presented in Section 4.1.2.1 pertains to the system clipboard that is used in performing copy/cut and paste operations.

3.6. Preventing illegal information flow

With few exceptions, existing access control mechanisms share fundamental weaknesses. One well known weakness is the inability to prevent the "leakage" of data to unauthorized principals through malware, or malicious or complacent user actions. To illustrate this weakness, assume the following three privileges: (u_1, r, o_1), (u_1, w, o_2), and (u_2, r, o_2). Considering these privileges alone, it is impossible to determine if u_2 can read the content of o_1. Under one scenario, u_1 can read and subsequently write the contents of o_1 to o_2. Even if the enforcement of policy were predicated on "trust in users", we must all but assume the existence of malware that can easily thwart the policy. This threat exists because, in reality, users do not perform operations on objects, but under a user's capabilities, processes perform operations on the content of objects. Therefore, a program executed by u_1, can read the contents of o_1, and without u_1's further action or knowledge, write those contents to o_2. Note that one cannot prevent this leakage even by adding a negative privilege, $\neg(u_2, r, o_1)$.

The importance of preventing inappropriate leakage of data (often called confinement) has been recognized as early as the 1970s, with the establishment of the Bell and LaPadula security model [9] and the Mandatory Access Control (MAC) policy [1]. Although MAC compliant systems go beyond traditional DAC and RBAC products in preventing inappropriate leakage of data, these systems are limited to multi-level security, and are not general enough to support numerous other policy objectives that also depend on confinement. Without this general support, commercially available products are arguably incapable of enforcement of a wide variety of policies, to include some instances of RBAC, e.g., "only doctors can read medical records", ORCON and Privacy [7], e.g., "I know who can currently read my data or personal information", or conflict-of-interest [2], e.g., "a user with knowledge of information within one dataset cannot read information in another dataset".

Not all advanced policies pertain to the prevention of data leakage to unauthorized principals. One such policy is Separation of Duty (SoD). While RBAC is noted for its support of SoD principles, real-world products enforce only the simplest forms (e.g., static and dynamic-based SoD), and as we later discuss, with great trepidation. Simon and Zurko, in their seminal paper on SoD [10], describe history-based SoD as the most accommodating form of SoD, subsuming the policy objectives of these other simpler forms. Other history-based policies pertain to two person control, workflow, and conflict-of-interest, but regardless of their importance in deterring fraud or combating other threats, enforcing history-based policies in a static privilege management environment is difficult or even impossible.

To address some of the above issues, a number of extensions to the commonly deployed access control frameworks have been proposed [11–13]. We feel this approach is flawed, amounting to further construction on a weak foundation (in one respect or another), to only address a specific problem. Others have proposed policy specification languages that provide a means of mapping policy onto existing access control mechanisms. Paramount among these languages is Ponder [8]. Although Ponder may be rich in its expression, the enforcement of policy is limited by the context that is made available by the underlying mechanisms.

The PM addresses these and other policy issues through its ability to dynamically alter its policy state based on a rich set of events pertaining to processes successfully accessing protected object contents. The PM prevents leakage of sensitive data to unauthorized principals by first recognizing the reading of sensitive information by a process and subsequently constraining that process or its user from writing to objects accessible to those unauthorized principals. This approach is general enough to support, with a reasonable degree of assurance, a large variety of policies that depend on the absence of leakage. Separation of duty and other history-based policies can be supported through a similar approach.

4. Example policy configurations

In this section, we demonstrate the PM's ability to express and enforce the policy objectives of RBAC, Chinese Wall, MAC and DAC models. It is important to note that when we say that the PM is able to express or enforce the policy objectives of a particular model we are not necessarily suggesting that we emulate the specific rules or relations of the model. Instead, the PM is able to implement the same policy objective of the model through a specific configuration strategy of its prescribed data sets and relations and its reference mediation function.

4.1. RBAC

Over the past decade and half, a number of RBAC models and RBAC extensions have been proposed [3–5]. In defining RBAC requirements, we refer to the RBAC standard [4], considered to be the culmination of the prominent RBAC models and implementations of the day.

4.1.1. RBAC specification

A major objective of RBAC is to streamline authorization management over identity-based access control schemes by defining roles as relations between users and capabilities. (The PM's notion of a capability is often referred to as permission in RBAC). These relations are achieved by assigning users to roles on one side and assigning capabilities to roles on the other side. By assigning a user to a role, that user acquires the capabilities that are assigned to the role.

Another important RBAC feature is the ability to define a role hierarchy, i.e., an inheritance relation between roles, whereby senior roles acquire the capabilities of their juniors. By assigning a user to a role, the user is also (indirectly) associated with the capabilities of that role's junior roles.

In addition to these administrative features, standard RBAC provides features to address Separation of Duty (SoD). SoD is a security principle used to formulate multi-person control policies, to reduce the likelihood of the occurrence of fraud, by requiring that two or more different people be responsible for the completion of a sensitive task or set of related tasks. Although the SoD principle predates RBAC [14], SoD is commonly defined in terms of roles and role relations. The RBAC standard includes two types of relations for the enforcement of separation of duties – static separation of duty (SSD) and dynamic separation of duty (DSD). SSD relations place constraints on the assignments of users to roles, whereby membership in one role may prevent the user from being a member of another role, and thereby presumably forcing the involvement of two or more users in performing a sensitive task that would involve the capabilities spread over both roles. Dynamic separation of duty relations, like SSD relations, limit the capabilities that are available to a user, while adding operational flexibility, by placing constraints on roles that can be activated within a user's sessions. As such, a user may be a member of two roles in DSD, but unable to execute the capabilities that span both roles within a single session.

4.1.2. PM configuration of RBAC

The PM retains and in many respects exceeds the administrative and policy objectives of RBAC. To help illustrate these points, consider the PM assignments depicted in Fig. 3. In this figure, Doctor, Intern, and Consultant are roles represented by instances of user attributes, where Doctor is assigned to Intern. Also included in the configuration are objects ($o_1 \ldots o_7$) that are assigned to object attributes (e.g., o_1 and o_2 are assigned to Med_Records). User attributes are also assigned to operation sets, and operation sets are assigned to object attributes. For example, Doctor $\rightarrow \{w\} \rightarrow$ Med_Records. Finally, all users, objects, and user and object attributes are mapped to the single policy class, RBAC. For reader's convenience, in Table 1, we list the set of privileges that can be derived from this configuration by applying the definition at the end of Section 3.2.

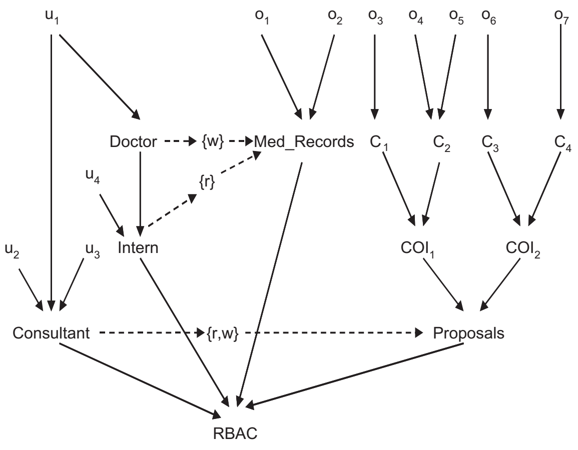

Fig. 3. Example RBAC assignment configuration.

4.1.2.1. Roles. User attributes indeed share the semantics of an RBAC role; by assigning a user to a user attribute, the user is indirectly associated with capabilities via the user attribute. For instance through the $u_1 \rightarrow$ Doctor assignment, u_1 is associated with the capabilities (w, o_1) and (w, o_2). Note the existence of the privileges (u_1, w, o_1) and (u_1, w, o_2) in Table 1. While RBAC is well noted for its administrative efficiency in associating users with capabilities, PM offers even greater efficiency as a consequence of its operation set and object attribute abstractions. That is, for each $ua \rightarrow ops \rightarrow oa$ relation, where $ua \in UA, ops \in 2^{OP}, oa \in OA, ua$ and any user assigned to ua is associated with capabilities equal to the number of operations in ops times the number of objects in oa. Under RBAC, capabilities are directly and individually assigned to roles. Furthermore, through these same relations, PM allows for a similarly efficient association of objects with access entries (e.g., o_2 is associated with (u_1, r)←and (u_1, w) $via o_2 \rightarrow$←Med_Records), while RBAC offers no similar semantics.

4.1.2.2. Role hierarchies. With regard to role hierarchies, the PM offers semantics similar to RBAC through user attribute to user attribute assignments. With respect to Fig. 3, in addition to u_1 having the capabilities to write to the objects in Med_Records through the $u_1 \rightarrow$←Doctor assignment, u_1 also has the capabilities to read the objects in Med_Records through the Doctor →← Intern assignment. In addition, the PM provides for the inheritance

Table 1
The list of derived privileges as a consequence of the assignments depicted in Fig. 3.

$(u_1, r, o_1), (u_1, w, o_1), (u_1, r, o_2), (u_1, w, o_2), (u_1, r, o_3),$
$(u_1, w, o_3), (u_1, r, o_4), (u_1, w, o_4), (u_1, r, o_5), (u_1, w, o_5),$
$(u_1, r, o_6), (u_1, w, o_6), (u_1, r, o_7), (u_1, w, o_7), (u_2, r, o_3),$
$(u_2, w, o_3), (u_2, r, o_4), (u_2, w, o_4), (u_2, r, o_5), (u_2, w, o_5),$
$(u_2, r, o_6), (u_2, w, o_6), (u_2, r, o_7), (u_2, w, o_7), (u_3, r, o_3),$
$(u_3, w, o_3), (u_3, r, o_4), (u_3, w, o_4), (u_3, r, o_5), (u_3, w, o_5),$
$(u_3, r, o_6), (u_3, w, o_6), (u_3, r, o_7), (u_3, w, o_7), (u_4, r, o_1), (u_4, r, o_2)$←

of access entries between object attributes (not depicted in Fig. 3), while again RBAC offers no semantics in this regard.

4.1.2.3. Separation of duty. Although RBAC SSD and DSD relations offer some advancement in control over identity-based systems, security issues remain. To illustrate this point, assume that a conflict-of-interest would arise if a single user were able to execute capability (op_1, o_1) and capability (op_2, o_2). Under RBAC, these capabilities could be assigned to different roles (say r_1 and r_2) and an SSD relation could be imposed on those roles and thus prevent any user from being simultaneously assigned to both roles. However, while any user u, assigned to r_1, would be prevented from executing (op_2, o_2) through denial of membership to r_2, nothing in the SSD relation prevents (op_2, o_2) from being assigned to some role r_3 and u being assigned to r_3. Now assume an RBAC environment where r_1 and r_2 are in DSD. Again, nothing prevents capability (op_1, o_1) and/or capability (op_2, o_2) from being assigned to some r_3 where r_3 is not considered in any DSD relation. Also, if a user is able to activate r_1 and r_2 in different sessions, either concurrently or sequentially, that user could execute capability (op_1, o_1) and capability (op_2, o_2).

The PM is able to meet the policy objectives of SSD and DSD, while alleviating these security issues. Again, assume that a conflict-of-interest would arise if a single user were able to execute capability (op_1, o_1) and capability (op_2, o_2). Now, consider the following PM event pattern/response relations:

(1) **when** process p performs (op_1, o_1) **do** create $u\_deny(process\_user(p), \{op_2\}, \{o_2\})$;
(2) **when** process p performs (op_2, o_2) **do** create $u\_deny(process\_user(p), \{op_1\}, \{o_1\})$.

Through relations (1) and (2) any process that successfully executes (op_1, o_1) would effectively deny the user of the process the ability to successfully execute (op_2, o_2) in the future and vice-versa. As such, a sensitive task consisting of (op_1, o_1) and (op_2, o_2) would require the independent actions of a minimum of two users to complete. Furthermore, this separation would hold independent of any privilege configuration (erroneous or otherwise), and independent of the sessions under which any process executed a capability of concern.

Perhaps the most operationally flexible and comprehensive form of SoD is history-based [10]. Under history-based SoD, if a user performs an operation on an object, that user can't perform a second operation (the same or different operation) on the same object. A simple example is a user both requesting and approving a purchase order. This form of SoD can easily be realized by the PM. For instance consider the following relation:

(3) **when** process p performs $(op_1, o_1)\leftarrow$**do** create $u\_deny(process\_user(p), \{op_2\}, \{o_1\})\leftarrow$

that specifies, if any user performs op_1 on object o_1, that user can no longer perform op_2 on o_1.

4.1.3. Preventing data leakage in RBAC

RBAC is not designed to prevent unauthorized leaking of data. For example, with respect to Fig. 3, the RBAC policy specifies that doctors and interns can read medical information, and this suggests to many that only doctors and interns can read medical information. Under this configuration, nothing prevents u_1 from reading or copying the contents of an object in Med_Records and writing or pasting it to an object in Proposals, and thus enabling u_2, who is not a Doctor, the ability to read medical information. Even if we were to trust doctors not to perform such actions, a malicious process acting on u_1's behalf could read medical information and write it to any object in Proposals without u_1's knowledge.

In order to prevent such a leakage, consider the following PM event-response relations

(4) **when** process p performs (r, o) where $o \rightarrow^{+}$Med_Records**do** create $p_deny(p, \{w\}, \neg$Med_Records$)$.

Relation (4) will prevent a single process from reading contents of any medical record (e.g., o_1) and subsequently writing it to any object outside the Med_Records container (e.g., o_3). Even with the existence of relation (4), two processes could cooperate in leaking data. With respect to the specific case of the copy/paste operation, one process could read data, e.g., a medical record, followed by a copy/paste operation from the memory of the first process to the memory of the second process followed by the second process writing the data to another object, making the data available to users that are not authorized to read the data. To prevent this method of leakage, assume the following relation is added to our configuration:

(5) **when** copy object o **do** assign clipboard to attributes of o.

Relations (4) and (5) together prevent copying of an object in Med_Records and the subsequent pasting of its contents into an object that is not in Med_Records. That is, the copy operation would place the clipboard into Med_Records, according to relation (5), and according to relation (4) any subsequent process that reads from the clipboard (e.g., paste) would be prevented from writing to any object (e.g., o_3) that is not in Med_Records.

As shown below, the same technique used to enforce confinement in RBAC can be applied to enforce confinement in context of any other model to include meeting the policy objectives of MAC.

4.2. Mandatory Access Control

In this section, we describe the policy objectives of MAC and present a PM configuration that meets these objective.

4.2.1. MAC specification

The objective of the MAC [9] security policy is to prevent the unauthorized reading of classified information. Traditionally, this policy objective has been specified and often implemented in terms of the simple security property, *-property (also referred to as the confinement property), and tranquility property of the Bell and LaPadula [9] security model, hereafter referred to as BLP. Under BLP, security levels, organized under a dominance relation, are assigned to subjects (users and their processes) and objects. We say that security level x dominates security level y if x is greater than or equal to y. The simple security property specifies that a subject is permitted read access to an object only if the subject's security level dominates the object's security level, and the *-property specifies that a subject is permitted write access to an object only if the object's security level dominates the subject's security level. Indirectly, the *-property prevents the transfer of data from an object of a higher level to an object of a lower classification. The security objective of these two rules is to prevent the direct and indirect reading of information at a level higher than the user's level. As commonly implemented, the security level of a process takes on the level of the session for which it belongs. The security level of a session (usually established at session creation time) can take on any single security level dominated by the clearance level of its user, but once assumed must remain fixed for the duration of the session. This condition is referred to as the tranquility property.

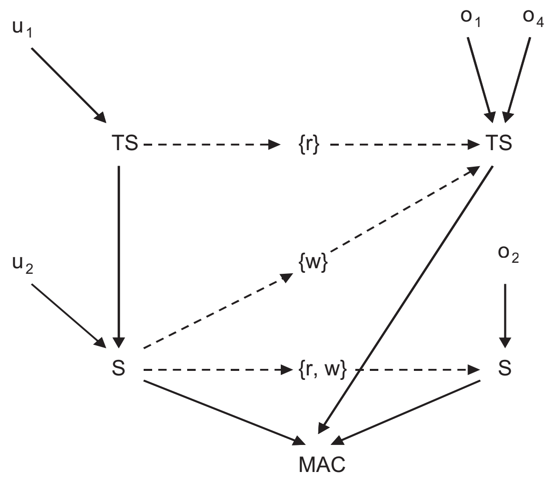

Fig. 4. Example MAC assignment configuration.

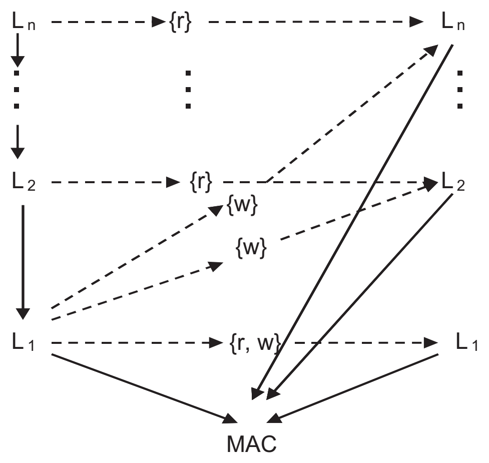

Fig. 5. A general MAC configuration.

The tranquility property serves two purposes. First it associates a process with a security level. Second it prevents, for example a process from reading top-secret data, storing the data in memory, switching its level to secret, and writing the contents of its memory to a secret object.

4.2.2. PM configuration of MAC

Fig. 4 is an example that illustrates PM assignment relations that serve as the basis for meeting the security objectives of BLP. Fig. 4 assumes top-secret dominates secret. It further specifies that users cleared to the levels of top-secret and secret are, respectively, assigned to the TS and S user attributes, and objects that are classified at the top-secret and secret levels are, respectively, assigned to the TS and S object attributes. With respect to these assignment relations, users (and their processes) that are assigned to TS are only able to perform read operations on objects classified at the levels top-secret and secret. Users (and their processes) that are cleared secret are only able to perform read operations on objects classified at the level secret, thus showing support for the security objectives of the simple security property. For the readers' convenience, Table 2 lists the set of privileges that can be derived from the assignment relations depicted in Fig. 4.

However, under these assignment relations, a user like u_1 for example, could read top-secret data and subsequently write that data to a secret object. To prevent this leakage assume the following two event pattern/response relations:

(6) **when** process p reads $o \rightarrow^{+}\mathrm{TS}$ **do** create $p\_deny(p, \{w\}, \neg\mathrm{TS})$;

(7) **when** process p reads $o \rightarrow^{+}\mathrm{S}$ **do** create $p\_deny(p, \{w\}, \neg(\mathrm{S} \cup \mathrm{TS}))$.

Relation (6) specifies that once a process successfully reads a top-secret object, the process can only write to objects that are in the TS container. Similarly, relation (7) specifies that once a process successfully reads a secret object, the process can only write to objects that are in S or TS containers.

Under this configuration, a process with its user cleared to a particular level (say top-secret), can read objects at levels at or below the user's clearance level (i.e., top-secret or secret). However, once a process has read data at a particular level (say top-secret), that process can no longer write to objects below that particular level (i.e., secret). These observations demonstrate adherence to the security objectives of the simple security, the *-property, and the tranquility property of the BLP security model, and thus the MAC Policy. Fig. 5 illustrates a construction strategy for the attributes and assignment relations of a MAC policy with n security levels $L_1, \ldots, L_n$ where L_{k+1} dominates L_k for $k = 1$ to $n-1$. The obligation relations that pertain to this construction are:

when process p reads $o \rightarrow^{+} L_n$ **do** create $p\_deny(p, \{w\}, \neg L_n)$;
when process p reads $o \rightarrow^{+} Ln-1$ **do** create $p\_deny(p, \{w\}, \neg(L_n \cup L_{n-1}))$;
...
when process p reads $o \rightarrow^{+} L_1$ **do** create $p\_deny(p, \{w\}, \neg(L_n \cup L_{n-1} \cdots \cup L_1))$;

Table 2
Derived privileges from the MAC assignments configuration of Fig. 4.

$(u_1,r,o_1),(u_1,w,o_1),(u_1,r,o_2),(u_1,w,o_2),(u_1,r,o_4),$ $(u_1,w,o_4),(u_2,w,o_1),(u_2,r,o_2),(u_2,w,o_2),(u_2,w,o_4)\leftarrow$

It is important to recognize that our PM MAC configuration does not only meet the policy objectives of MAC, it does so with greater user transparency than that of common BLP implementations. Under these implementations, all processes running in a session are labeled at the same level as the session. This condition has the effect of imposing undue restrictions on the session user. For example, a user cleared to top-secret in a top-secret session cannot write to a secret object. Under the PM framework and our MAC configuration, restrictions are dynamically imposed at the process level, thus allowing the user, through multiple processes, the ability to execute his/her full range of authorized capabilities within a single session. A user can read top-secret data, and subsequently write to secret data through different processes running in the same session. Also, note that the MAC PM configuration does not specify unclassified user or object attributes, yet, relations (6) and (7) ensure that classified information cannot be leaked to unclassified objects (which are all objects not included in S or TS).

For the non-hierarchical component of a MAC policy, there exist efficient PM configurations. Given the space limitations, we favored the specification of other policy configurations.

4.3. Discretionary Access Control

In this section we describe the control objectives of DAC, and present a PM configuration that meets these objectives.

4.3.1. DAC specification

Central to Discretionary Access Control (DAC) [1] are the concepts of object ownership and control. The owner of an object is a user that possesses administrative capabilities to grant/revoke other users or groups of users, access to the object (i.e., creation and modification of access entries for the object). The owner of an object is typically the user that created the object. Control pertains to the set of administrative capabilities that enables the creation and modification of access control entries associated with owned objects. Control may also pertain to the transfer of ownership to another user. Ownership of an object also implies capabilities to read and write the object.

4.3.2. PM configuration of DAC

The PM offers a number of strategies for the configuration of DAC policies. Under our strategy a user's identity is represented through a user attribute that specifies the name or identity of the user (i.e., the user in question is the only user assigned to this user attribute). We call this attribute the "name attribute", or the "identity attribute". Similarly, group identities are represented as user attributes that contains only the users that are members of that group. Fig. 6, which partially illustrates a PM DAC configuration, where the user attribute "Alice Smith" is user alice's name attribute, while the "DAC users" user attribute represents the collection of all users included in the DAC policy class.

User's ownership and capabilities over an "owned" object can be specified under this PM configuration by placing the object in a container specially created for that user. We refer to this container as the user's home. In Fig. 6, the object attribute "alice home" denotes the home container for user alice. The creation of a user's home is accompanied by setting up three categories of capabilities for the user: (a) capabilities to access objects in the home container; (b) capabilities to perform administrative operations on the elements and relations comprising the home container for the organization of its contents (e.g., object attribute to object attribute assignments, creation of new object attributes); and (c) capabilities to transfer ownership or grant/revoke other users' access to the objects inside the home container. The user, his/her home container and the capabilities (a), (b), and (c) could be conveniently created through a single administrative command – *create_dac_user* (*user id*, *user name*). Typically, under DAC a user initially obtains ownership and control over an object as a consequence of object creation. This can be achieved by defining an event-response relation where the event is the object creation and the response is the assignment of the new object to the user's home container.

Using the policy configuration described above, transferring the ownership of an object to another user may be achieved by assigning the object to the other user's home container and optionally deleting its assignment to the original owner's home. Note that the transfer requires the privilege to assign objects from the original owner home to another user's home container.

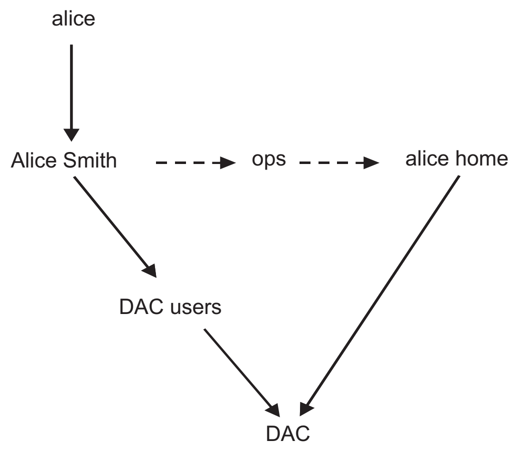

Fig. 6. A partial DAC configuration.

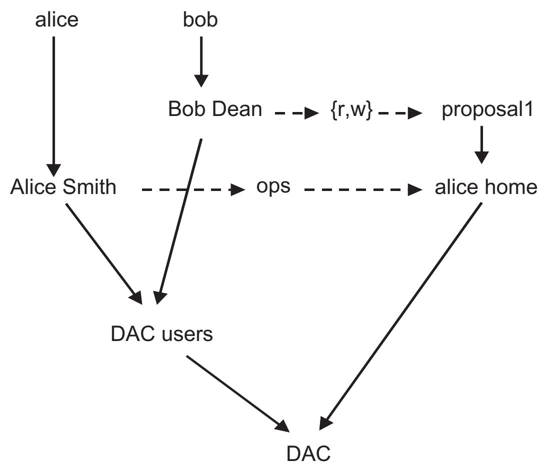

Fig. 7. Alice grants bob read/write access to proposal1.

Granting another user or group of users access to an object o may be achieved by creation of the assignment $g \rightarrow \{r, w\} \rightarrow o$ where g is a user attribute that represents the other user or group of users in the DAC users. Fig. 7 shows how alice could grant user bob read/write access to one of her objects by using such assignments to bob's name attribute "Bob Dean".

4.4. Chinese Wall policy

While the purpose of SoD is to reduce the likelihood of fraud, the purpose of the Chinese Wall policy, as modeled by Brewer and Nash [2], is to address conflict-of-interest issues related to business practices. Consultants or advisors are naturally given access to proprietary information to provide a service for their clients. When a consultant gains access, for example to the competitive practices of two banks, the consultant gains knowledge amounting to insider information that can undermine the competitive advantage of one or both institutions or can be used for personal profit. The objective of the Chinese Wall policy and its associated model is to identify and prevent user accesses as well as the possibility for the flow of information that can give rise to such conflicts.

4.4.1. Chinese Wall specification

Under the Brewer and Nash model, company sensitive information is categorized into mutually disjoint conflict-of-interest categories (COIs). Each company belongs to only one COI and each COI has two or more member companies. The membership within a COI includes companies whereby a consultant obtaining sensitive information regarding one company would constitute a conflict-of-interest if the consultant were to obtain sensitive information in regard to another. Several COIs may co-exist. For example, one COI may pertain to Banks, while another COI may pertain to Energy Companies. Brewer–Nash defines two rules, one for reading and one for writing:

- ←Read Rule: Subject s can read object o only if:
  - o is in the same company dataset as some object previously read by s, or
  - o belongs to a COI class for which s has yet to read an object.
- ←Write Rule: Subject s can write object o only if

– s can read o under the read rule, and
– No object can be read within a different company dataset than the one for which write access is requested.

It is important to recognize that the Brewer–Nash rules consider subjects to include both users and the processes that are acting on behalf of the users, and the rule for writing takes into consideration the possibility of a Trojan horse that can leak sensitive data outside a given company dataset.

4.4.2. PM configuration of Chinese Wall

The Brewer–Nash model begins with the recognition of objects each belonging to a single company dataset. Our strategy and example of the enforcement of the Chinese Wall policy begins with the select assignment of objects $o_3 \cdots o_7$ to company data sets represented by containers C1 $\cdots$ C4 in the context of the RBAC assignment relations of Fig. 3. As per the Brewer–Nash model, company datasets are further categorized into conflict-of-interest (COI) classes. Our configuration, vis. Fig. 3, includes two such COIs, COI1 and COI2, that are assigned to the container Proposals, and the Consultant role that has read and write access to all objects in Proposals.

In addition to the assignment configuration of Fig. 3, we meet the objectives of the Chinese Wall policy through the following event pattern/response relation:

(8) when process p performs (r, o), where $o \to^+$Proposals **do**
create $u\_deny(process\_user(p), \{r\}, \{oa_2 \cap \neg oa_1\})$,
create $p - deny(p, \{r, w\}, \{\neg oa_1\})$,
where $o \to oa_1 \to oa_2 \to$ Proposals.

Relation (8) is valid under the assumption that for every object o, such that $o \to^+$Proposals, there exists unique object attributes oa_1 and oa_2 such that $o \to oa_1 \to oa_2 \to$ Proposals. Indeed, in our configuration oa_1 is a company dataset, and oa_2 is a COI. Relation (8) specifies that whenever a process performs a read operation on an object contained in Proposals, a user-deny and a process deny relation are created. The user-deny relation prohibits the user of the process that had read the object o, the capability to subsequently read objects that are contained in the COI of o but not in the company data set of o. For example, if a process with u_2 as its user reads object o_5, u_2 could still read any object in C2 (e.g., o_4 and o_5), but can no longer read an object in COI1 that is not in C2 (e.g., o_3 in C1). Also, under this example, u_2 can still read any object in Proposals that is not in C1. Note that this aspect of relation (8) meets both requirements of the Brewer–Nash rule for reading. The created process deny relation prohibits a process that had read an object o in Proposals the capability to write to any object outside of company dataset of o, or to read an object from outside company dataset of o. Note that although the user of the process may be able to read objects outside the company dataset for which it has write access, the process would be prevented from doing so. This second aspect of relation (8) meets both requirements of the Brewer–Nash rule for writing.

5. Combining policies

In this section we describe the PM's natural ability to control user and process access to objects under the combination of two or more policies. Consider the combination of the RBAC and MAC policy configurations as specified above. First assume the assignment configurations of Figs. 3 and 4. Again, for the readers' convenience we list the derived privileges that can be derived through combination of the assignment relations of these figures in Table 3.

Note that not all users or objects are in both policy classes. For instance, u_3, o_3, o_5, o_6 and o_7 are only in RBAC, but all objects in MAC are also in RBAC. Regardless of the number of policies under which an object is protected, the same rules for reference mediation apply. That is, a process access request $\langle op, o\rangle_p$ is granted iff there exists a PM privilege (u, op, o)←where $u = process\_user(p)$, and (op, o) has not been denied for either u or p. Reference mediation begins with the determination of a PM privilege. See definition of a PM privilege in Section 3. Note that with respect to the combination of configurations of Figs. 3 and 4, the triple (u_1, w, o_1) is a PM privilege, because (1) o_1 is in both RBAC and MLS, (2) both Doctor and S are u_1's user attributes, where Doctor is in RBAC and S is in MLS, (3) TS and Med_Records are object attributes of o_1, where Med_Records is in RBAC and TS is in MLS, and (4) the operation set containing w is assigned to Doctor and Med_Records, and S and TS. The triple (u_2, r, o_3) is also a privilege because o_3 is only in RBAC and (u_2, r, o_3)←is a PM privilege in RBAC. In contrast, (u_2, r, o_2) is not a privilege, because o_2 is in both RBAC and MAC, Consultant is the only attribute of u_2 in RBAC, but Consultant cannot read o_2 in RBAC.

Table 3
The list of privileges that can be derived though the combination of assignment relations of the RBAC policy assignment relations depicted in Fig. 3 and the MAC policy assignment relations depicted in Fig. 4.

$(u_1, r, o_1), (u_1, w, o_1), (u_1, r, o_2), (u_1, w, o_2), (u_1, r, o_3),$
$(u_1, w, o_3), (u_1, r, o_4), (u_1, w, o_4), (u_1, r, o_5), (u_1, w, o_5),$
$(u_1, r, o_6), (u_1, w, o_6), (u_1, r, o_7), (u_1, w, o_7), (u_2, r, o_3),$
$(u_2, w, o_3), (u_2, w, o_4), (u_2, r, o_5), (u_2, w, o_5), (u_2, r, o_6),$
$(u_2, w, o_6), (u_2, r, o_7), (u_2, w, o_7), (u_3, r, o_3), (u_3, w, o_3),$
$(u_3, w, o_4), (u_3, r, o_5), (u_3, w, o_5), (u_3, r, o_6), (u_3, w, o_6),$
$(u_3, r, o_7), (u_3, w, o_7)$←

Implicit to our definition of a privilege is the notion of need-to-know. Under need-to-know restrictions, even if a user has all the necessary official approvals (such as a clearance) to access certain information under one policy, the user would not be given access to the information unless the user has a specific need to know; that is, access to the information must be necessary for the conduct of one's official duties. Although the need-to-know principle has its origins in limiting access to classified information under the combination of both MAC and DAC, this principle can and often does apply to combinations of other policy classes as well.

For instance, although a user may be a Doctor, and thus can read patient medical records under an RBAC policy, the user is denied the ability to read a particular medical record, unless the user can read the medical record by virtue of being assigned to a particular ward, under an organization based policy. Another explanation as to why (u_2, r, o_2) is not a privilege is that, although u_2 is cleared secret and o_2 is secret, o_2 is a medical record that can be accessed by Doctors or Interns and u_2 is neither. In other words, u_2 does not have a need-to-know. On the other hand, regarding $(u_1, w, o_1), o_1$ is classified TS and is a medical record, and u_1 is cleared to top-secret and is also a doctor, therefore u_1 can write o_1 under the need-to-know principle.

Another interesting property is that not all objects need to be included in a policy class that supports an overall policy, in order to be included in the scope of control of the policy. For the remainder of this section, assume the existence of the assignment relations of Figs. 3 and 4, and relations (4)–(8) are also in place. Although o_3 (presumed unclassified) is not included in the MAC policy class, o_3 is included in MAC's scope of control. Indeed, if process p reads object o_1 (top-secret), p is then denied, through relation (6), write access to o_3. It is also the case that if process p first reads object o_3, p can write to o_1. In general, relations (6) and (7) prevent a process from writing to an object at a level less than the level of any object that it had previously read. Relation (5) with (6) and (7) prevent the transfer of classified information through a copy and paste operation to an unclassified object. In contrast, in keeping with BLP, for a user to access an object, the object (classified or unclassified must be labeled. In this respect, the PM's is more efficient in the embodiment of the MAC policy than that of BLP.

We now consider the combination of the MAC and Chinese Wall policies, where the Chinese Wall policy has no policy class of its own. Note that o_4 is classified TS, and is also in the company dataset C2, along with o_5 that is not classified. According to the Chinese Wall policy once a user has read an object in a company dataset that user is free to read and write to other objects in the same company dataset. Indeed, in accordance with our configuration, once a user has read o_4, that user can read o_5, and in fact that user can write to o_5, using a different process than the one used for reading o_4. In addition, a user can copy the contents of o_5 and paste it into o_4, but at the same time, the user cannot copy any portion of o_4 and paste it into o_5. If there existed another object also classified TS, in a different company dataset, but in the same COI as o_4 the contents of o_4 cannot be written or pasted into the object. This is precisely the behavior one would expect under the combinations of MAC and Chinese Wall.

It is important to note that all properties that pertain to policy combinations are achieved without the need for any further configuration beyond those configurations of the constituent policies.

6. Related work

Although the PM may exhibit features similar to those of other access control frameworks, as noted in Section 1, the PM is neither a new access control model, nor an extension or adaptation of any existing access control model or mechanism. The PM is a redefinition of access control enforcement for providing a unifying framework to support a wide range of policies under a single mechanism. The PM is not unique in this pursuit of generalizing access control and offering policy flexibility. Therefore, we do not review the many access control models and their extensions proposed in the literature, but only review the attempts to generalizing access control.

One partial solution to meet general policy needs is an OASIS' standard eXtensible Access Control Markup Language (XACML) [6] and the Ponder policy specification language [8]. XACML describes both a policy language and an access control decision request/response language (both encoded in XML). The policy language describes general access control requirements. The request/response language allows for queries to ask whether a given action should be allowed and interpret the result. A similarity between XACML and the PM is their ability to afford policy combinations with respect to privileges. One drawback of XACML is that it does not specify or enforce policies that pertain to processes in isolation to their users, thereby disallowing the specification and enforcement of a wide variety of related policies. Another drawback of XACML is that its Policy Decision Point is stateless, which place limitations on the policies that can be specified and enforced. Although XACML includes the concept of an obligation, it is not used to alter the state of the policy. Ponder is declarative object oriented language for specifying security and management policy for distributed object systems. Under Ponder policy is expressed through the use of their language and is enforced through mappings onto various access control mechanisms, thus separating policy from the implementation of the system. The PM is a logical and complete "machine" in that the configuration of its relations expresses policies which are enforced through a reference mediation function that is part of the machine. The policies that are enforceable by Ponder are limited by the underlying mechanisms. For example, we have shown that placing prohibitions through obligations on processes to be an instrumental component in the enforcement of confinement policies. Although Ponder includes the concept of obligations, we doubt that Ponder can afford similar control.

Although products that protect objects under an MLS policy traditionally also protect these same objects under a need-to-know policy, such products afford these policy combinations through the deployment of two separate mechanisms, one in support of the MLS policy and the other in support of the need-to-know policy. PM is different in this regard in its ability to enforce multiple arbitrary policies through the application of a single mechanism.

Another partial solution would be to use various configurations of Role-Based Access Control relations to simulate Mandatory Access Control and Discretionary Access Control policies. This was demonstrated by Osborn et al. [15], using the RBAC96 model [5]. One drawback to this approach is that Osborn et al. applied a series of obligation constraints in the configuration of these policies that can only exist in theory, and are not specified in the RBAC96 model. Although RBAC96 alludes to a variety of possible constraints, an implementation of RBAC96 would not necessarily include the specialized obligation constraints that were applied in the construction of the MAC policy. Simulating MAC in [15] requires for each object the explicit creation of two capabilities (using PM terminology), each assigned to a role. In PM each object is naturally assigned to an object attribute that represents the object's classification. Although we did not present a strategy for addressing categories, while Osborn did, we can point out that our requirements are linear in the number of represented categories. A drawback of their strategy for simulating DAC requires the creation of a multitude of roles that would exceed the number of objects in the system. In our configuration for each user, the user is assigned to his/her user name attribute, and user capabilities to afford DAC responsibilities are achieved by assigning the user name attribute to a predefined operation set that is assigned to an object attribute (the home of a user) that contains all objects for which a user has DAC control and ownership. This suggests that the PM's MAC and DAC configurations are more natural embodiment than that of RBAC's configuration.

As an extension to the RBAC model, Crampton [11] proposes the use of "blacklists" as a means of enforcing history-based SoD constraints. This use of blacklists is analogous to the PM's concept of dynamically creating a user-deny relation in the context of an event-response relation. Although Crampton is able to enforce a wide variety of SoD policies, his method is limited in comparison to the PM. Through the event-response relations, the PM is capable of not only dynamically creating user-deny relations, but also process deny relations, and in general can create and delete any type of relation through any series of administrative operations.

Another partial solution is an early and incomplete description of the PM [16]. A drawback of this solution is the limitation and inefficiency in specifying and enforcing policy. The proposed framework required the costly computation and activation of a set of user attributes for a set of processes running in a session, in order to gain access to a resource. Further drawbacks include the lack of control at the individual process level, the lack of constraints on users and processes, and the inability to dynamically alter the policy state of the machine in support of the specification and enforcement of policy.

The PM presented in this paper is more closely related to the meta-model for specifying and enforcing a generic access control model. Recently, the need for a meta-model has been argued by Ferraiolo and Atluri [17]. Responding to these discussions, Barker [18] has proposed a meta-model of access control and presents a logic language for describing the meta-model. He shows how arbitrary access control policy requirements can be represented in the proposed meta-model by making small changes to the core concepts of the meta-model. He demonstrates how a range of existing access control models can be viewed as instances of this meta-model. Several similar Proposals have been made in the desire to define a general, declarative framework for specifying a wide range of access control policies including the RT family of role-trust models [19], FAF language [20] and SecPAL [21]. While these Proposals provide a unifying model, the PM additionally offers a mechanism for implementing the unifying model.

7. Conclusions and vision

In this paper, we have presented an access control framework, referred to as the Policy Machine (PM), that fundamentally changes the way in which access control mechanisms are developed. The PM framework aids in developing high assurance systems because of three fundamental reasons. First, it minimizes the amount of code to be trusted as it decouples security enforcement from the host system. Second, it allows the precise policies of the resource owners independent of that offered by the vendors, which therefore does not require unsupported policies implemented as separate application code. Third, it prevents illegal information flow in commonly used DAC and RBAC models.

Both users of the PM as well as the developers of the host systems enjoy several benefits. From the user perspective, PM can be employed as a general purpose protection machine as one mechanism can configure many types of access control policies. PM offers a large library of policies available for immediate configuration. Since access control is enforced at the enterprise level encompassing the different OS and applications in use within the organization, and the users need to login only to the PM, it naturally provides interoperability and single sign-on. Moreover, it offers fine-grained, flexible and comprehensive protection to the enterprise resources. Since the security policy enforcement is separated from that of the functional components such as the OS and application software, the part that needs to be trusted is smaller, which results in higher operational assurance. Additionally, there is no policy enforcement or decision making done at the application. As shown in Section 4, PM can render many Trojan horse attacks harmless. Although not discussed in this paper, many applications such as e-mail and workflow management can be implemented by merely configuring the PM as they can be seen as extensions to access control. As a result, sensitive data can be protected from being leaked to unintended recipients. Currently, this can be accomplished only through the trusted behavior of the users. Under PM, an unauthorized principal cannot gain access to a sensitive file even if it was e-mailed to him as the e-mail file sharing is mediated via PM. Additionally, sharing of data across other secondary storage devices (e.g., hard-drives, memory sticks) can all me controlled under PM.

From the vendor perspective, vendors need not produce different versions or change their systems to accommodate the policy de jour, and there is no need to cater to special needs of different user communities. The vendor products need not implement the access control decision modules and maintain or manage access control data. The same applies to any application being used within the organization. Specifically, application developers need to focus on implementing only the functionality of their application rather than the access control.

The big winner would be the customer that gets to implement their individual and precise policy requirements through acquiring PM components and the translation of those requirements into a PM data configuration. To facilitate this translation, standard configurations for a variety of policies can be made available as a library of parameterized policy configurations. This reduces the burden on administrators in specifying and configuring policies.

In addition to providing a high assurance access control enforcement framework, PM can be utilized to implement a number of applications such as e-mail and workflow management. Although not normally considered in the realm of access control, both e-mail and workflow management falls into this application class, in that e-mail affords the sharing of information through the discretionary distribution of messages and attachments, and workflow prescribes sequences of user accesses to documents. These can be simply realized through PM as they can be viewed as extensions of access control.

References

[1] DoD Computer Security Center, Trusted Computer System Evaluation Criteria, December 1985.

[2] D. Brewer, M.J. Nash, The Chinese Wall security policy, in: Proceedings of the IEEE Symposium on Security and Privacy, 1989, pp. 206–214.

[3] D. Ferraiolo, R. Kuhn, Role-Based Access Control, in: Proceedings of the 15th NIST-NCSC Computer Security Conference, Washington, DC, 1992, pp. 554–563.

[4] D.F. Ferraiolo, R. Sandhu, S. Gavrila, D.R. Kuhn, R. Chandramouli, Proposed NIST standard for role-based access control, ACM Transactions on Information and System Security (TISSEC) 4 (3) (2001) 224–274. <http://doi.acm.org/10.1145/501978.501980>.

[5] R. Sandhu, E. Coyne, H. Feinstein, C. Youman, Role-Based Access Control models, IEEE Computer (1996) 38–47.

[6] X. TC, Oasis eXtensible Access Control Markup Language (XACML) Version 2.0.

[7] R. Graubart, On the need for a third form of access control, in: Proceedings of the National Computer Security Conference, 1989, pp. 296–304.

[8] N. Damianou, N. Dulay, E. Lupu, M. Sloman, The ponder policy specification language, in: Proceedings of the Workshop on Policies for Distributed Systems and Networks, 2001, pp. 18–39.

[9] D. Bell, L. LaPadula, Secure Computer Systems: Unified Exposition and MULTICS Interpretation, Technical Report MTR-2997, The Mitre Corporation, Bedford, MA, March 1976.

[10] R. Simon, M. Zurko, Separation of duty in role based access control environments, in: Proceedings of the New Security Paradigms Workshop, 1997.

[11] J. Crampton, Specifying and enforcing constraints in role based access control, in: Proceedings of the ACM Symposium on Access Control Models and Technologies, 2003, pp. 43–50.

[12] R. Ferrini, E. Bertino, Supporting RBAC with XACML + OWL, in: Proceedings of the ACM Symposium on Access Control Models and Technologies, 2009.

[13] Z. Mao, N. Li, H. Chen, X. Jiang, Trojan horse resistant discretionary access control, in: Proceedings of the ACM Symposium on Access Control Models and Technologies, 2009.

[14] J. Saltzer, M. Schroeder, The protection of information in computer systems, Proceedings of the IEEE 63 (9) (1975) 1278–1308.

[15] S. Osborn, R. Sandhu, Q. Munawer, Configuring role-based access control to enforce mandatory and discretionary access control policies, ACM Transactions on Information Systems Security 3 (2) (2000) 85–106.

[16] D. Ferraiolo, S. Gavrila, V. Hu, R. Kuhn, Composing and combining policies under the policy machine, in: Proceedings of the ACM Symposium on Access Control Models and Technologies, 2005, pp. 11–20.

[17] D. Ferraiolo, V. Atluri, A meta model for access control: why is it needed and is it even possible to achieve? in: Proceedings of the 13th ACM Symposium on Access Control Models and Technologies, 2008, pp. 153–154.

[18] S. Barker, The next 700 access control models or a unifying meta-model? in: Proceedings of the 14th ACM Symposium on Access Control Models and Technologies, 2009, pp. 187–196.

[19] N. Li, J. Mitchell, W. Winsborough, Design of a role-based trust-management framework, in: Proceedings of the 2002 IEEE Symposium on Security and Privacy, 2002, p. 114.

[20] S. Jajodia, P. Samarati, M.L. Sapino, V.S. Subrahmanian, Flexible support for multiple access control policies, ACM Transactions on Database Systems 26 (2) (2001) 214–260.

[21] M. Becker, C. Fournet, A. Gordon, Design and semantics of a decentralized authorization language, in: Proceedings of the 20th IEEE Computer Security Foundations Symposium, 2007, pp. 3–15.

David Ferraiolo is the manager of the Systems and Network Security group at NIST and is the lead for the Cyber Security Working Group in support of the Portfolio Manager for Standards in the Directorate for Science and Technology at the Department of Homeland Security. He has 23 years of experience in computer and communications security, serving both the government and private industry.

Vijay Atluri is a professor of computer information systems and a research director at the Center for Information Management, Integration and Connectivity at Rutgers University. In addition, she is a computer scientist with NIST's Information Technology Laboratory, Computer Security Division, Systems and Network Security Group.

Serban Gavrila has been with VDG, Inc., since 1995 and with NIST since March 2007. For NIST, he works on projects related to computer security. Those projects include development of formal specifications and implementation of enterprise-wide access control management systems, and security policy management for enterprise-issued hand-held devices.