

(TS) and save the latter object, it is easy to show (applying the same kind of reasoning) that she succeeds.

[0136] In another example, in the case when the copy and paste operations are performed in different processes, user alice issues a request to read `mrec1` (TS) in a process p_1 . The reference mediation function grants the request. At the successful completion of the read operation, a deny relation (p_1 , $\{w\}$, \neg TS) is added to the policy configuration as specified by the event-response relation (1). Also, an event-response

[0137] (3.2) p_1 creates object \Rightarrow assign new object to TS

[0138] is generated according to (3). Next, alice issues a request to read `mrec2` (S) in a new process p_2 . The reference mediation grants the request. At the successful completion of the read operation, a deny relation (p_2 , $\{w\}$, \neg S_TS) is added to the policy configuration as specified by the event-response relation (2). Also, an event-response:

[0139] (4.2) p_2 creates object \Rightarrow assign new object to S

[0140] is generated according to (4).

[0141] alice may try to copy some information from object `mrec1` (top-secret) to object `mrec2` (secret) and save the latter. To copy the information from object `mrec1` to the clipboard, the process p_1 first creates an object that represents the clipboard. According to (3.2) and (4.2) the new object is assigned to TS. Second, the process p_1 actually copies the information from `mrec1` to the clipboard and a “copy object” event is generated. According to relation (5), the clipboard object is assigned to all attributes of `mrec1`. The fact that the clipboard object is already assigned to TS does not matter. Hence, the clipboard object becomes assigned to TS and Med Records.

[0142] Next, alice pastes the clipboard content to the `mrec2` object in process p_2 . The paste action starts with a read operation from the clipboard object, which is classified TS. According to the event-response relations (1), the system **20** generates the deny relations (p_2 , $\{w\}$, \neg TS). The clipboard content is pasted into the `mrec2` object. Finally, alice tries to save (write) the `mrec2` object. Because `mrec2` is not contained in TS, the deny relation (p_2 , $\{w\}$, \neg TS) prevents the current session from saving `mrec2`.

[0143] Another example shows that 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 that only doctors and interns can read medical information. Under this configuration, nothing prevents bob from copying the contents of `mrec3` and pasting it into the object `project1`, which can be read by charlie who is not a Doctor or Intern. It should be noted that a malicious process acting on bob’s behalf could also read medical information and write it to `project1` without bob’s knowledge.

[0144] To prevent this unlawful leakage, the system **20** can apply the approach that was used to prevent leakage under MLS to the context of RBAC. Consider the following event-response relation:

[0145] (6) read “Med Records” object \Rightarrow create deny(current process, $\{w\}$, \neg “Med Records”).

[0146] Relation (6) will prevent bob using a single process from reading contents of any medical record (e.g., `mrec3`) and subsequently writing it to any object outside the Med Records container (e.g., `project1`).

[0147] In a scenario where bob copies data from `mrec3` and pastes it to `project1` in different processes, relation (5) assigns the clipboard object to the Med Records container. The second process for the paste operation reads the clipboard object, and according to the relation (6) the system **20** generates a deny relation that prevents bob from writing (saving) `project1`.

[0148] In another example, under Discretionary Access Control (DAC), the user who creates an object is called the object “owner” and controls users’ capabilities on that object, based on the users’ or user groups’ identities. The capabilities that the owner controls include operations on the object’s content (e.g., read/write/execute), as well as operations that change the object’s access control policy (e.g., transfer ownership of the object or grant/revoke users’ access to the object).

[0149] The system **20** can be programmed to achieve the objectives of DAC policies. For example, a user’s identity can be represented through a user attribute that specifies the name of the user and which has that user as its only member (i.e., the user in question is the only user assigned to this user attribute). The attribute may be called a “name attribute”. Similarly, a group identity could be specified as a user attribute that contains only the users that are members of that group. In FIG. 8, which partially illustrates a system **20** configured to achieve a DAC objective, the user attribute “Alice Smith” is user alice’s name attribute, while the “DAC users” user attribute represents the group of all users included in the DAC policy class.

[0150] User’s ownership and capabilities over an “owned” object can be specified under this configuration by placing the object in a container specially created for that user. We refer to this container as the user’s home. For example, the object attribute “alice home” denotes the home container of user alice. The creation of a user’s home must be accompanied by setting up three categories of capabilities for the user: (a) capabilities to access the content of the objects contained in the home container; (b) capabilities to perform administrative operations on the contents of the home container (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 the system **20** 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.

[0151] 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 permission to assign objects from the original owner home to another user’s home container.

[0152] Granting another user or group of users access to an object o may be achieved by the owner 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. 9 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”. Other configuration strategies exist as well.

[0153] In another example, an additional feature of the system **20** is the capability to establish a library of policy configurations. The principle is that an administrator does not need to configure policy from scratch. Once a policy (say DAC) has been defined and tested by security experts, the policy can be made available for importation and instantiation. Policy configuration can also be parameterized, providing opportunities for customization. For example, with