JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, no. 3, March-April 2004

L^nRBAC: A Multiple-Levelled Role-Based Access Control Model for Protecting Privacy in Object-Oriented Systems

Shih-Chien Chou, Department of Computer Science and Information Engineering, National Dong Hwa University, Taiwan

Abstract

Role-based access control (RBAC) is useful in information security. It is a super set of discretionary access control (DAC) and mandatory access control (MAC). Since DAC and MAC are useful in information flow control (which protects privacy within an application), RBAC can certainly be used in that control. Our research reveals that different control granularity is needed in different cases when controlling information flows within an application. An information flow control model should thus simultaneously offer different levels of control granularity. We designed a multiple-leveled RBAC model to offer multiple levels of control granularity, in which a level of RBAC controls a level of granularity. We called the model L^nRBAC (n-leveled RBAC), which offer the following features: (1) it allows different control granularity in different cases, (2) it solves the covert channel problems caused by abnormal program stopping, (3) it adapts to dynamic object state change, (4) it controls method invocation through argument sensitivity (5) it allows purpose-oriented method invocation, (6) it controls write access precisely, and (7) it avoids Trojan horses. We implemented a prototype for L^nRBAC and evaluated it. This paper presents L^nRBACL.

1 INTRODUCTION

Privacy protection within an application is essential for an application that manages sensitive data. The protection can be achieved by information flow control, which prevents information in high security levels from flowing to subjects in low security levels (i.e., the control block non-secure information flows). Many information flow control models have been developed, among which some applied mandatory access control (MAC) [Bell 1976] [Denning 1976] [Denning 1977], some applied discretionary access control (DAC) [Samarati 1997] [Ferrari 1997], some applied the label approach

Cite this article as follows: Shih-Chien Chou: "L^nRBAC: A Multiple-Levelled Role-Based Access Control Model for the Protecting Privacy in Object-Oriented Systems", in *Journal of Object Technology*, vol. 3, no. 3, March-April 2004, pp. 91-120.
http://www.jot.fm/issues/issue_2004_03/article2

[Myers 1998] [Myers 2000] [McCollum 1990] [McIlroy 1992], and some applied role-based access control (RBAC) [Izaki 2001]. Our research applies RBAC. Below we briefly introduce RBAC and then back to the discussion of RBAC on privacy protection.

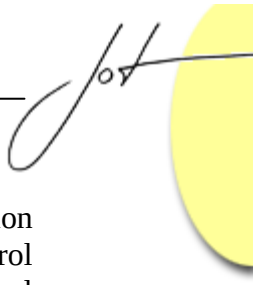
A RBAC model is primarily composed of users, roles, sessions, permissions, various assignment relationships among the previous components, and constraints. A role is a collection of permissions [Sandhu 1996a]. Within a session, a user possesses the permissions of the role he plays. Role assignment is based on user responsibilities. That is, the role assigned to a user should possess permissions to facilitate finishing the user's responsibility. When a user finishes his responsibility, the role assignment will be removed, which results in revoking permissions from the user. Users can change role during his responsibility if necessary. This facilitates providing no extra privileges (i.e., enforcing the need-to-know principle) [Sandhu 1996a]. A major advantage of RBAC is that permissions are bound to roles instead of users. With this, dynamic adjustment of user permissions can be achieved through role assignment.

The research in [Osborn 2000] [Sandhu 1996a] proved that RBAC is a super set of DAC and MAC. Since DAC and MAC are useful in information flow control, RBAC can certainly be used in that control. Currently we identify the research in [Izaki 2001] applied RBAC in that control. We involved for years in the research of applying RBAC to control information flows within object-oriented systems. From the research, we experienced a problem related to the control granularity of security as described below.

In the original design of RBAC, users are human beings or agents [Sandhu 1996a]. Permissions are access rights from users to objects (an object can be a table in a relational database or an object in an object-oriented database). In this case, the control granularity is detailed to the table/object level. This level of control granularity is insufficient in controlling information flow within an application, because information within an object-oriented system is generally stored in object attributes or method variables (we collectively call them *variables* in the rest of this paper). In this regard, the control granularity should detail to the variable level (and therefore information in variables can be protected). Contradicting to this control granularity is detailing the control granularity to objects [Samarati 1997] or methods [Izaki 2001] [Yasuda 1997]. Below we use a man/woman example to explain why detailing the granularity to objects or methods is insufficient.

Suppose that a man and a woman may be strangers, friends, or husband and wife. If they are strangers, no information flow among them is allowed. If they are friends, they can read each other's general information, such as name, address, e-mail address, and so on. If they are married, a marriage certificate should exist. In this case, they can read and write each other's general information, and can read each other's personal information, such as birthday, health condition, and so on. Moreover, they can read the information of their marriage certificate, which cannot be accessed by persons other than the couple.

In the above example, if the control granularity is detailed to objects only, we can only control the woman that can be accessed by a man, but cannot control the woman's methods and variables that can be accessed by the man. In this regard, if a man "m1" can



access his female friend “w1”, “m1” can access both the general and personal information of “w1” by invoking the methods of “w1”. With this control granularity, the control requirements mentioned above cannot be achieved. On the other hand, if the control granularity is detailed to methods only, we can only control the methods of a man/woman that can be invoked by a woman/man, but cannot control the variables that can be accessed by a method. In this regard, if a man’s method “m1.md1” can invoke his female friend’s method “w1.md1”, both the general and personal information of “w1” can be offered to “m1” through “w1.md1”. This control granularity, again, cannot achieve the control requirements mentioned above.

According to the description above, detailing control granularity to variables is necessary. However, there are cases that coarser grained of control granularity are needed. For example, as described in the man/woman example, no information flow is allowed among strangers, whereas information flows among friends or between husband and wife are allowed. In this case, a control mechanism that details the control granularity to objects is needed to determine the legality of information flows among objects. As another example, a man can invoke methods of his female friends to handle information flows between friends. Moreover, he can invoke other methods of his wife to handle information flows between husband and wife. In other words, a woman’s methods that can be invoked by her friends and those that can be invoked by her husband are generally different. In this case, a control mechanism that details the control granularity to methods should be available to determine the legality of method invocations (this feature is also called purpose-oriented method invocation in [Yasuda 1997]).

In addition to the above cases of control granularity, an information flow control model should better offer a much coarser grained of control granularity to solve the covert channel problems [Focardi 1997] induced by abnormal program stopping. We explain this case below. When a program is abnormally stopped, the operating system will dump the memory space used by the program to a file. The debugger then helps programmer to debug the program using the dumped file. If the dumped file is accessed by an unauthorized user or application, private information within the program may be leaked. To prevent this, a model should control the human beings or applications that can access a file or run a program. In this regard, a control granularity details to just programs or files is needed.

As a summary, four types of control granularity should be simultaneously offered by an information flow control model. In the past years, we developed an RBAC-based information flow control model called OORBAC [Chou in press]. OORBAC details the control granularity to variables. Moreover, it incorporates complex mechanisms to detail control granularity to objects and methods. According to the experiences of using OORBAC, the control in OORBAC is too complicated. Moreover, OORBAC fails to solve the problems induced by covert channels. We thus revised OORBAC. The basic consideration of this revision is using multiple-leveled RBACs, in which one level of RBAC offers one level of control granularity. An information flow should fulfill every level of RBAC for the flow to be secure. Currently, the revised model is composed of the following four levels of RBAC.

1. Level 0 RBAC (L0RBAC). It solves the covert channel problems mentioned above by detailing the control granularity to programs and files.
2. Level 1 RBAC (L1RBAC). It details the control granularity to objects, which determines the legality of information flows among objects.
3. Level 2 RBAC (L2RBAC). It details the control granularity to methods, which determines the legality of method invocations.
4. Level 3 RBAC (L3RBAC). It details the control granularity to variables, which determines the security of information flows among variables.

We call the revised model L^nRBAC (n-leveled RBAC), which offers the following features:

1. It allows different control granularity in different cases
   The need for this feature is the motivation of LnRBAC, in which different level RBAC provides different control granularity.
2. It solves the covert channel problems caused by abnormal stopping of program.
   This feature is offered by L0RBAC. In the level, users are human beings or programs, and permissions are access rights from human beings or programs to programs or files. With the permissions, unauthorized human beings and applications cannot access dumped files when a program is abnormally stopped.
3. It adapts to dynamic object state change.
   During program execution, objects may be dynamically instantiated or deleted. Moreover, object relationships may be dynamically established or removed. We call a snapshot of objects and object relationships at a time point an object state. Information flow control model should adapt to dynamic object state change. For example, if initially a man and a woman are strangers, no information flow is allowed between them. When they become friends at a time point, information flows for friends should be allowed between them. When they get married at another time point, information flows for husband and wife should be allowed between them. This feature is achieved by L1RBAC and L2RBAC.
4. It allows purpose-oriented method invocation.
   The research in [Yasuda 1997] identified the needs for purpose-oriented method invocation. With this, the legality of method invocations should be ensured. This consideration is correct because methods may be in different secure levels and therefore should be protected independently [Varadharajan 1990]. For example, if a man and a woman are friends, he can invoke a method that retrieves her general information but cannot invoke a method that retrieves her personal information. This feature is achieved by L2RBAC.
5. It controls method invocation through argument sensitivity.
   This feature is useful, although the models we surveyed did not emphasize it. For example, suppose a man “m1” can change his wife’s general information using “m1.others_new_general_info” as an argument. Then, using other variables such

as the man's general information as an argument in the invocation should be denied. The rationale is that different variables carry different information and therefore should be used for different purposes. This feature is achieved by L3RBAC.

6. It controls write access precisely.
   According to our survey, most existing models paid poor attention to write access control. They merely obeyed the "no write down" rule [Bell 1976] to control write access. Nevertheless, write access is destructive and therefore should be controlled precisely. Otherwise, data corruption may occur according to intentional or accidental mistakes. We propose that only the data sources trusted by a variable can write the variable. This feature is achieved by L3RBAC.
7. It avoids Trojan horses.
   Avoiding Trojan horses [Myers 1998] [Myers 2000] is the basic feature that should be offered by every information flow control model. This feature is achieved by the join operation [Myers 1998] [Myers 2000].

This paper presents L^{n}RBAC. Since every level RBAC of L^{n}RBAC is an adaptation of RBAC96, we introduce RBAC96 briefly before describing L^{n}RBAC. We also describe the features of L^{n}RBAC.

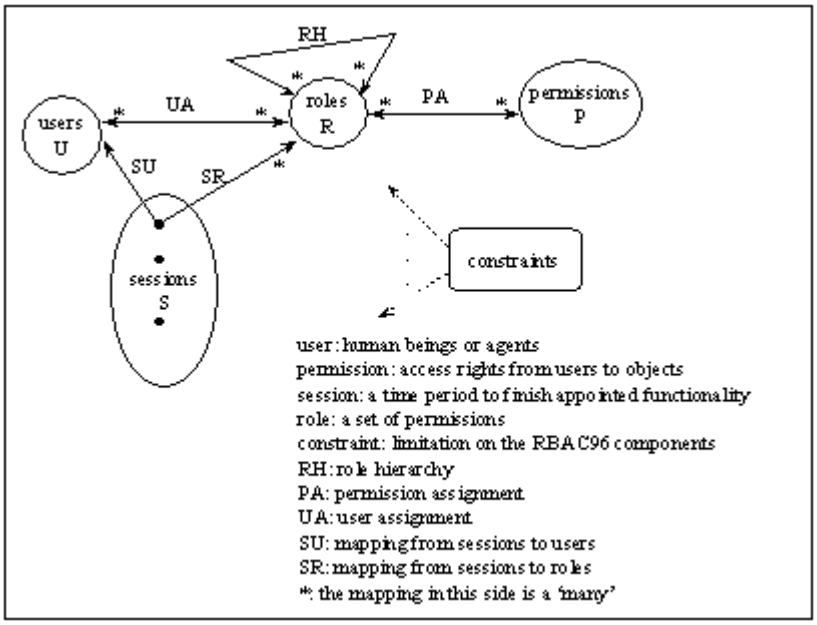


Figure 1: RBAC 96

2 RBAC96

This section briefly introduces RBAC96. Details of it can be found in [Sandhu 1996a]. Figure 1 shows RBAC96, which is composed of the following components:

a) A set of permissions (P). A permission approves a mode of access on an object.
b) A set of roles (R). A role is composed of a set of permissions.
c) A many-to-many permission to role assignment (PA). A permission may be assigned to multiple roles and a role may be assigned multiple permissions.
d) A partially ordered role hierarchy (RH). Roles are structured using the "$\geq$" relationship. If a relationship "x $\geq$ y" exists, "x" possesses all the permissions of "y".
e) A set of users (U), which is a human being or an agent. Users play roles. A user playing a role possesses the permissions of the role.
f) A set of sessions (S). A user establishes a session during which he plays one or more roles.
g) A many-to-many user to role assignment (UA). A user may play many roles within a session, and may establish multiple sessions simultaneously. Moreover, multiple users may play the same role. In addition, users can change role to facilitate providing no extra privileges in a session [Sandhu 1996a].
h) A function that maps a session to a single user (SU). Using the function, users in a session can be identified.
i) A function that maps a session to a set of roles (SR). Using this function, the permissions of a session can be identified.
j) A collection of constraints limiting the model elements.

3 L^{n}RBAC

The most challenge work in designing L^{n}RBAC is adapting to dynamic object state change (remember that an *object state* is *a snapshot of objects and object relationships at a time point*). We use the man/woman example mentioned in section 1 and the object states in Figure 2 to explain this. Figure 2(a) depicts two men and two women. It also shows a marriage relationship among the man "m1", the woman "w1", and the certificate "cer1", and shows three friendship relationships between men and women. Figure 2(b) depicts one newly added man "m3" and one newly added woman "w4". It also shows that "m1" and "w1" get divorced and then become strangers (in this case, the certificate "cer1" should be deleted). Moreover, the figure shows various marriage and friendship relationships between men and women. Figure 2(c) shows that "w3" is past away. Since "m2" does not marry another woman, the marriage between "m2" and "w3" is still legal and therefore the certificate of the marriage, "cer3", need not be deleted.

In the object state shown in Figure 2(a), information flows among "m1", "w1" and "cer1" should obey the rules for a marriage because a "married" relationship exists among them. In addition, information flows between "m1" and "w2", those between "m2" and "w2", and those between "m2" and "w3" should obey the rules for friends because a "friends" relationship exists between the pairs of man and woman. Moreover, information flows between "m2" and "w1" are disallowed because they are strangers (i.e., no relationship exists between them). The allowed and disallowed information flows in Figures 2(b) and 2(c) will be different from those in Figure 2(a) because of different object states. As a summary of the above description, the allowed and disallowed information will change according to object state change.

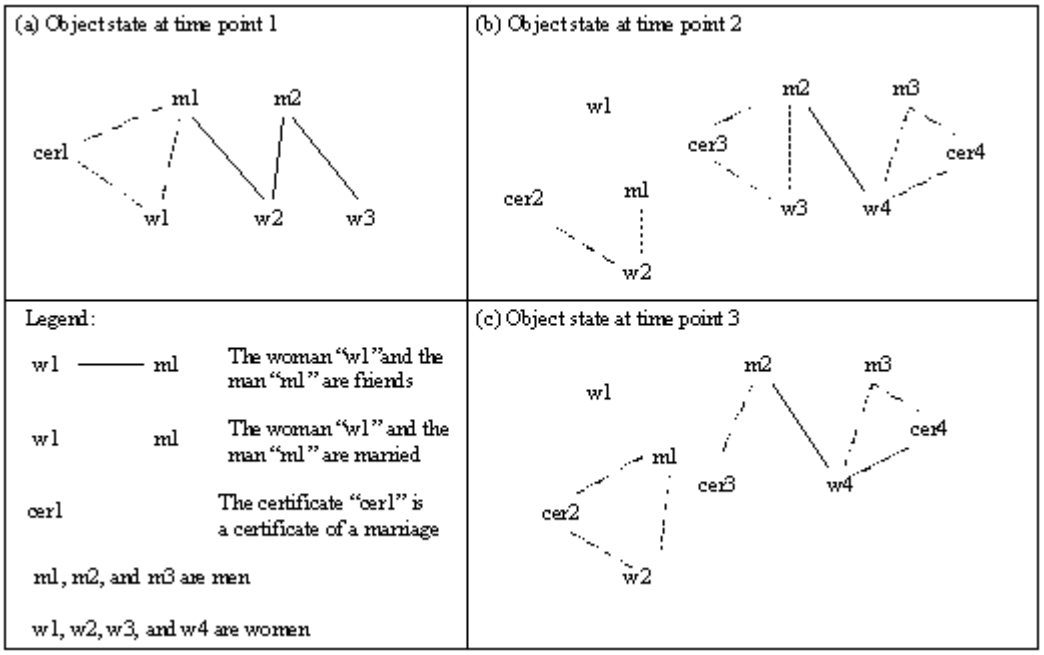


Figure 2: Object state change

Our research reveals that objects relationships [Rumbaugh 1999] can be used to determine whether an information flow is secure or not. We thus use them to regulate information flows among objects. We call a relationship an *association*, and give the following definition:

> *An association exists among classes if information may directly flow among the instances of the classes. Each association is associated with a security policy for class instances to obey. If multiple security policies must be obeyed by class instances, more than one association should be defined among the classes, in which an association enforces a security policy.*

In the above definition, an association is a relationship among classes, which can be instantiated to link objects of the classes. Objects linked by an association *coexist in an*

association group (AG) according to the association. For example (see Figure 2(a)), "m1", "w1", and "cer1" coexist in an AG according to the association "married". Direct information flows are allowed among objects coexisting in an AG but disallowed among objects not coexisting in an AG. Since information cannot directly flow among sessions, an AG should be established for objects that may communicate. Although information cannot directly flow among sessions, it may indirectly flow among AGs. For example, if "obj1" and "obj2" are in an AG and "obj2" and "obj3" in another, information from "obj1" may indirectly flow to "obj3" via "obj2".

L^{n}RBAC model

This section defines the four level RBACs in L^{n}RBAC, in which only L0RBAC does not use the concept of association. L0RBAC regulates the access rights from human beings or programs to programs or files. It is defined below:

L0RBAC = *(U0, S0, P0, R0, RH0, PA0, UA0, SU0, SR0)*, in which "U0" is the set of users; "S0" is the set of sessions; "P0" is the set of permissions; "R0" is the set of roles; "RH0" is the set of role hierarchies; "PA0" is the set of permission to role assignment; "UA0" is the set of user to role assignment; "SU0" is the set of functions that map a session to users, and "SR0" is the set of functions that map a session to roles. The definition of L0RBAC's components are shown below:

U0 = *{u | u is a human being or a program}*

S0 = *{s | s is a time period during which a person runs a program or a program accesses a file}*

P0 = *{(u, o, a) | $u \in U0$, o is a program or a file, $a \in \{r, w, e\}$, and u is allowed to access o in which the allowed access is indicated by a}*. P0 defines the access rights from users to programs or files, in which an access may be a read, a write, or an execute.

R0 = *{r | r is a set of permissions}*

RH0 = *{$r0 \geq r1$ | r0, $r1 \in R0$ and r0 possesses all permissions of r1}*

PA0 = *{(r, p) | $r \in R0$ and $p \in P0$ and p is assigned to r}*

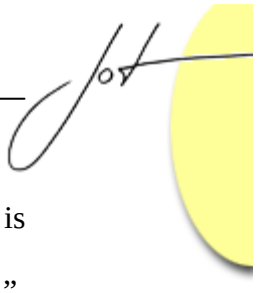
UA0 = *{(u, r) | $u \in U0$ and $r \in R0$ and u is assigned to r}*

SU0 = *{f | f is a function, f(s) = U in which $s \in S0$ and $U \subseteq U0$, and every user u in U is in the session s}*

SR0 = *{f | f is a function, f(s) = R in which $s \in S0$ and $R \subseteq R0$, and every role r in R is within the session s}*

L1RBAC determines whether information flows between two objects are allowed. It is defined below:

L1RBAC = *(C, A1)*, in which "C" is the set of classes in an application and "A1" is the set of associations in the application. Remember that an association can be



instantiated to produce AGs for objects to coexist. The component "A1" is defined below:

A1 = *(U1, S1, P1, R1, RH1, PA1, UA1, SU1, SR1, CT1)*, in which "U1", "S1", "P1", "R1", "RH1", "PA1", "UA1", "SU1", "SR1" are similar to "U0", "S0", "P0", "R0", "RH0", "PA0", "UA0", "SU0", "SR0" in L0RBAC. Moreover, "CT1" is the constraints of L1RBAC. Components in "A1" are defined below:

U1 = *{u | u is an object instantiated from a class, i.e., from a member of C}*

S1 = *{s | s is an AG according to an association, i.e., s is an instance of an association}*

P1 = *{(o0, o1) | o0, o1 ∈ U1 and information flows among o0 and o1 are allowed}*

R1 = *{r | r is a set of permissions}*

RH1 = *{r0 ≥ r1 | r0, r1 ∈ R1 and r0 possesses all permissions of r1}*

PA1 = *{(r, p) | r ∈ R1 and p ∈ P1 and p is assigned to r}*

UA1 = *{(u, r) | u ∈ U1 and r ∈ R1 and u is assigned to r}*

SU1 = *{f | f is a function, f(s) = U in which s ∈ S1 and U ⊆ U1, and every user u in U is in the session s}*

SR1 = *{f | f is a function, f(s) = R in which s ∈ S1 and R ⊆ R1, and every role r in R is within the session s}*

CT1 = *{ct | ct is a cardinality constraint or a modality constraint}*

From the above description, security policies of L1RBAC are embedded within associations. Note that L1RBAC also defines cardinality and modality constraints among classes [Pressman 2001].

L2RBAC determines whether an invocation between two methods is allowed. It is defined below:

L2RBAC = *(C, A2)*, in which "C" is the set of classes and "A2" is the set of associations. The component "A2" is defined below:

A2 = *(U2, S2, P2, R2, RH2, PA2, UA2, SU2, SR2)*, in which "U2", "S2", "P2", "R2", "RH2", "PA2", "UA2", "SU2", "SR2" are similar to "U0", "S0", "P0", "R0", "RH0", "PA0", "UA0", "SU0", "SR0" in L0RBAC. Components in "A2" are defined below:

U2 = *U1*

S2 = *S1*

P2 = *{(m1, m2) | m1 and m2 are object methods and m1 is allowed to invoke m2}*

R2 = *{r | r is a set of permissions}*

RH2 = *{r0 ≥ r1 | r0, r1 ∈ R2 and r0 possesses all permissions of r1}*

PA2 = *{(r, p) | r ∈ R2 and p ∈ P2 and p is assigned to r}*

UA2 = *{(u, r) | u ∈ U2 and r ∈ R2 and u is assigned to r}*

SU2 = *SU1*

SR2 = *{f | f is a function, f(s) = R in which s∈S2 and R⊆R2, and every role r in R is within the session s}*

L3RBAC determines whether an information flow is secure. It is defined below:

L3RBAC = *(C, A3)*, in which "C" is the set of classes and "A3" is the set of associations. The component "A3" is defined below:

A3 = *(U3, S3, P3, R3, RH3, PA3, UA3, SU3, SR3, DSOURCE)*, in which "U3", "S3", "P3", "R3", "RH3", "PA3", "UA3", "SU3", "SR3" are similar to "U0", "S0", "P0", "R0", "RH0", "PA0", "UA0", "SU0", "SR0" in L0RBAC. As to DSOURCE, it records the data source of a variable. For example, suppose the attribute "attName" is derived from the variable "var1" and "var2", and "var1" and "var2" are respectively written by the methods "mdx" and "mdy". Then, the DSOURCE of "attName" is the set "{mdx, mdy}" after the derivation. A DSOURCE is set empty initially. It will obtain contents during program execution through the join operation (see section 3.3). DSOURCEs facilitate controlling write access (see section 3.3). Components in "A3" are defined below:

U3 = *U1*

S3 = *S1*

P3 = *{(v, RACL, WACL) | v is a variable, RACL = {m | m is a method that is allowed to read v}, and WACL = {m | m is a method that is allowed to write v}}*

R3 = *{r | r is a set of permissions}*

RH3 = *{r0≥r1 | r0, r1∈R3 and r0 possesses all permissions of r1}*

PA3 = *{(r, p) | r∈R3 and p∈P3 and p is assigned to r}*

UA3 = *{(u, r) | u∈U3 and r∈R3 and u is assigned to r}*

SU3 = *SU1*

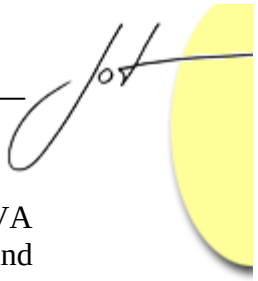
SR3 = *{f | f is a function, f(s) = R in which s∈S3 and R⊆R3, and every role r in R is within the session s}*

DSOURCE = *{f | f is a function, f(v) = {m | m is a method and m is a data source of v}, and v is a variable}*

Using L^{n}RBAC

We embedded L^{n}RBAC in the JAVA language to produce the L^{n}RBACL language. APPENDIX 1 shows the man/woman example mentioned in section 1 implemented in L^{n}RBACL. The object states shown in Figure 2 are implemented in the appendix. In the implementation, we suppose that the class "example", which contains the method "main", and the method "example.main" possesses every permission we needed because the class "example" is not the focus of this example.

The appendix shows that an L^{n}RBACL program is composed of two parts, namely the RBAC part and the original JAVA program. Moreover, the RBAC part consists of



L0RBAC through L3RBAC, and two non-JAVA statements are used in the JAVA program to define object states. They are “addAG” (line 15.4.4) to create an AG and “removeAG” (line 15.4.21) to remove an AG.

L0RBAC defines the permissions of three roles. The role “operator” is allowed to execute the program “a.exe” and “debugger.exe”. The role “program” is allowed to read the file “file1.dat” and write “file2.dat”. The role “debugger” is allowed to read “dump.core”. Here we suppose that the program in the appendix will be compiled into “a.exe”, the debugger is “debugger.exe”, and the operating system dumps the memory space used by “a.exe” to “dump.core” when “a.exe” is abnormally stopped. According to L0RBAC, only the debugger can access the dumped memory and only an operator can execute the debugger, this prevents the dumped file from being accessed by unauthorized persons or applications.

L1RBAC in APPENDIX 1 will be enacted when “a.exe” is executed (i.e., when L0RBAC initiates the session of executing “a.exe”). This level RBAC declares that man objects can access woman objects and vice versa under an AG according to the association “friends”. It also declares that man (woman) objects can access woman (man) objects and certificate objects under an AG according to the association “married”. Information flows among objects of the classes not appear in L1RBAC is not allowed. L1RBAC also declares the cardinality and modality constraint of an association. For example, line 3.1.1 indicates that a man can have multiple female friends (i.e., the cardinality is “*”) and a man need not have a female friend (i.e., the modality is “O”). As another example, line 3.3.3.1 indicates that a marriage should exist for a certificate (i.e., the modality is “M”) and one certificate can be associated with only one marriage (i.e., the cardinality is “1”).

L2RBAC in APPENDIX 1 will be enacted when L0RBAC initiates the session of executing “a.exe”. This level RBAC declares the allowed method invocations. For example, line 5.3.2.2 declares that the method “m1.change_others_general_infor” can invoke the method “w1.change_self_general_info” within an AG according to the association “married”. Here “m1” is a man and “w1” is a woman. The role hierarchy in line 5.3.1 says that a permission possessed by the association “friends” is also possessed by the association “married”.

L3RBAC in APPENDIX 1 will be enacted when L0RBAC initiates the session of executing “a.exe”. A permissions in this level RBAC declares the object methods that can read and write a variable. The permission related to a variable is an ACL of the variable, which composed of a RACL (read access control list) and a WACL (write access control list). See line 7.1.1.1 for an example, which defines the ACL of the variable “man.self_general_info” in the association “friends”. RACLs, WACLs, and DSOURCEs ensure secure information flows.

Information flow security in L^{n}RBAC

When executing an application, the operating system checks the information in L0RBAC to ensure that the execution is legal. During the execution of an application, the

corresponding L1RBAC, L2RBAC, and L3RBAC are created. To check the security of an information flow, we first check the type of information flow. If the flow is induced by method invocation, L1RBAC down to L3RBAC should be involved to check the flow's security. If the flow is not induced by method invocation (i.e., the information flow is within a method), only L3RBAC is invoked in the checking. Below we describe the use of the three levels of RBAC.

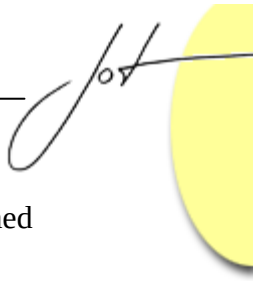
1. If an information flow is induced by a method invocation, e.g., "obj1.md1" invokes "obj2.md2", L1RBAC is first involved to check whether information flows between "obj1" and "obj2" are allowed. If the answer is negative, the information flow is non-secure.
2. If the above answer is positive, L2RBAC is involved to check whether the invocation from "obj1.md1" to "obj2.md2" is allowed. If the answer is negative, the information flow is non-secure.
3. If an information flow is induced by method invocation and both the above checking passes, the ACLs and DSOURCEs of arguments should be copied to the corresponding parameters. The copying is secure because a parameter receiving the value of an argument inherits the security level of the argument. Note that if an object is passed as an argument, the copying is bypassed because ACLs and DSOURCEs of the object's variables are already defined.
4. After the above copying, every information flow in the invoked method should fulfill the following secure flow conditions. The conditions are established using ACLs in L3RBAC based on the following assumption: (a) a value derived from the variables "var1", "var2", "varn", and so on is assigned to the variable "d_var", (b) the assignment appears in the method "md1", (c) the original ACL of "d_var" is "{RACLd_var; WACLd_var}", (d) the ACL of the ith variable that derives "d_var" is "{RACLvari; WACLvari}", and (e) the DSOURCE of "vari" is "DSOURCEvari".

First secure flow condition: $(RACL_{d\_var} \subseteq (RACL_{var1} \cap RACL_{var2} \cap ... \cap RACL_{varn})) \wedge (md1 \in (RACL_{var1} \cap RACL_{var2} \cap ... \cap RACL_{varn}))$

Second secure flow condition: $WACL_{d\_var} \supseteq (DSOURCE_{var1} \cup DSOURCE_{var2} \cup ... \cup DSOURCE_{varn} \cup \{md1\})$

The first secure flow condition controls read access. The requirement "$RACL_{d\_var} \subseteq (RACL_{var1} \cap RACL_{var2} \cap ... \cap RACL_{varn})$" requires that "d_var" must be the same restricted as or more restricted than "var1", "var2", "varn", and so on. The requirement "$md1 \in (RACL_{var1} \cap RACL_{var2} \cap ... \cap RACL_{varn})$" is necessary because the variables "var1", "var2", "varn", and so on are read by the method "md1".

The second secure flow condition controls write access. It requires that the data sources of "var1", "var2", "varn", and so on should be within "$WACL_{d\_var}$", because the data derived from the variables are written to "d_var". The requirement also requires that



the method "md1" must be within "$WACL_{d\_var}$" because the write operation is performed by the method.

After the derived data is assigned to the variable "d_var", the ACL of "d_var" should be changed by the join operation [Myers 1998] [Myers 2000]. This change is to avoid Trojan horses. We use the symbol " $\oplus$ " to represent the join operator. With join, "ACL_{d_var}" will be changed to be "$ACL_{var1} \oplus ACL_{var2} \oplus \ldots \oplus ACL_{varn}$" after the derived data is assigned to the variable "d_var". The join operation is defined below:

$$ACL_{var1} \oplus ACL_{var2} \oplus \ldots \oplus ACL_{varn} = \{RACL_{var1} \cap RACL_{var2} \cap \ldots \cap RACL_{varn} \; ; \; WACL_{var1} \cup WACL_{var2} \cup \ldots \cup WACL_{varn}\}$$

The join operation trusts less or the same readers. Therefore, join will not lower down security level. On the other hand, the operation trusts more writers. This is reasonable because a writer that can write a variable should be regarded as a trusted data source for the data derived from the variable. In addition to joining ACLs, the DSOURCE of "d_var" will be adjusted as follows:

$$DSOURCE_{d\_var} = DSOURCE_{var1} \cup DSOURCE_{var2} \cup \ldots \cup DSOURCE_{varn} \cup \{md1\}$$

"$DSOURCE_{d\_var}$" is set the union of "$DSOURCE_{var1}$", "$DSOURCE_{var2}$", "$DSOURCE_{varn}$", "{md1}", and so on. The union of the DSOURCEs is obvious because all data sources deriving the computation result should be considered data sources of the result. The method "md1" is also a data source because the computation result is *written* by "md1" to "d_var".

4 FEATURES

APPENDIX 1 to show that L^nRBAC does offer the feature mentioned in section 1. Note that avoiding This section use the man/woman example mentioned in section 1 and the corresponding code in Trojan horse can be achieved by the join operation (see [Chou in press]). Moreover, offering different control granularity in different cases is an implicit feature of L^nRBAC.

Solve the covert channel problems caused by abnormal stopping of program

This feature is achieved by L0RBAC. In APPENDIX 1, L0RBAC declares that the dumped file can be accessed by the debugger only and the debugger can be executed by the operator only. With this, the operating system can prevent unauthorized persons and applications from accessing the dumped file. This solves the covert channel problems caused by abnormal program stopping.

Adapt to dynamic object state change

This feature is achieved by L1RBAC and L2RBAC. For example, the statements between line 15.4.1 and 15.4.7 establish the object state of Figure 2(a). The statements between lines 15.4.8 through 15.4.12 are allowed because they pass the security requirements of the three levels of RBAC. The statement in line 15.4.14 is non-secure because "mm[2]" and "ww[1]" are strangers at that time point (i.e., they do not coexist in an AG). In this case, L1RBAC will block the statement. The statement in line 15.4.15 is also non-secure because "mm[2]" and "ww[3]" are within an AG according to the association "friends". In this association, there is no permission for "mm[2].get_others_personal_info" to invoke "ww[3].get_self_personal_info" (see lines 5.1 through 5.2). Therefore, the statement will be blocked by L2RBAC.

The object state in Figure 2(b) is established by the statements between lines 15.4.19 and 15.4.30. That in Figure 2(c) is established by the statements in line 15.4.43. Moreover, some secure and non-secure statements follow the establishment of the object states. To show that L^{n}RBAC adapts to dynamic object state change, let's check the statement in line 15.4.9 and that in line 15.4.40. The former statement is secure because in the object state of Figure 2(a), "mm[1]" and "ww[1]" are married. Nevertheless, the latter statement is non-secure because "mm[1]" and "ww[1]" get divorced in the object state of Figure 2(b). L2RBAC will screen out the latter statement. From the above description, L^{n}RBAC does adapt to dynamic object state change.

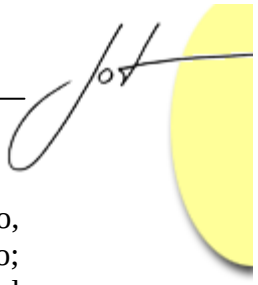
Allow purpose-oriented method invocation

This feature is achieved by L2RBAC. For example, line 5.3.2.2 in APPENDIX 1 shows that the method "change_others_general_info" of a man can invoke the method "change_self_general_info" of a woman if the man and the woman are within an AG according to the association "married".

Control method invocation through argument sensitivity

This feature is achieved by L3RBAC. We use an example to explain this. As described in section 1, a man "m1" can change the general information of his wife "w1". This change is accomplished by the method "m1.change_others_general_information" (line 9.8), which invokes the method "w1.change_self_general_info" (line 11.10) using the attribute "m1.others_new_generl_info" (line 9.8.2) as an argument. According to the ACLs in lines 7.3.1.3 and 7.3.3.1, both secure flow conditions are true in the above invocation (remember that DSOURCEs are initially empty). Therefore, the information flow induced by the above invocation is secure. Suppose another attribute of "m1", such as "m1.self_general_info" is used in that invocation, the information flow induced by the invocation will be non-secure. Let's trace the invocation below:

When "m1.change_others_general_information" (line 9.8) invokes the method "w1.change_self_general_info" (line 11.10) using the attribute "m1.self_generl_info" as



an argument, the ACL of the argument, which is "{m1.get_self_general_info, w1.get_others_general_info, m1.change_self_general_info; m1.change_self_general_info, w1.change_others_general_info}" (line 7.3.1.1), is copied to the parameter "new_general_info" of the method "w1.change_self_general_info". When executing the statement "self_general_info = new_general_info;" (line 11.10.1) within the method, the above ACL is compared with the ACL of "w1.self_general_info", which is "{w1.get_self_general_info, m1.get_others_general_info, w1.change_self_general_info; w1.change_self_general_info, m1.change_others_general_info}" (line 7.3.3.1). The comparison shows that the first secure flow condition is false and therefore the information is non-secure.

Control write access precisely

This feature is achieved by L3RBAC. We use an example to explain this. As described in section 1, a man "m1" can change the general information of his wife "w1". This change is accomplished by the method "m1.change_others_general_information" (line 9.8), which invokes the method "w1.change_self_general_info" (line 11.10). If the method "m1.get_others_general_info" (line 9.2)" tries to invoke the method "w1.change_self_general_info", the invocation will be blocked (i.e., the change is not allowed). The rationale is that the invocation fails to fulfill the second secure flow condition because the method "m1.get_others_general_info" is not within the WACL of the variable "w1.self_general_info" (line 7.3.3.1).

5 RELATED WORK

This section surveys related work according to the features mentioned in section 1. Covert channel problem is not discussed because we cannot identify a model that solves the problem.

The simplest information flow control approach is DAC. Since DAC fails to avoid Trojan horses, MAC [Bell 1976] [Denning 1976] [Denning 1977] was proposed. An important milestone of MAC is the model proposed by Bell&LaPadula [Bell 1976], which categorizes the security levels of objects and subjects. Information flows in the model follow the "no read up" and "no write down" rules [Bell 1976]. Bell&LaPadula's model was generalized into the lattice model [Denning 1976] [Denning 1977] (see [Sandhu 1993] for a survey of lattice models). In the typical lattice model proposed in [Denning 1976] [Denning 1977], a lattice $(SC, \rightarrow, \oplus)$ is constructed using "SC", which is the set of security classes, the symbol "$\rightarrow$", which is the "can flow" relationship, and the symbol "$\oplus$", which is the join operator. The "can flow" relationship controls information flows and the join operator avoids Trojan horses. Relationships between the features mentioned in section 1 and MAC are described below. First, MAC cannot detail the control granularity to different levels in different cases. The control granularity is decided by nodes in the lattice. For example, if nodes in the lattice are variables, the control granularity is detailed to variables. Second, MAC cannot adapt to dynamic object

state change because the lattice in a MAC is fixed during program execution. Therefore, object state change should be predicted before program execution. Third, purpose-oriented method invocation can be achieved if nodes in the lattice are object methods. Nevertheless, the control granularity will only be detailed to methods in this case. Fourth, controlling method invocation through argument sensitivity was not considered in the MACs we surveyed. Fifth, MAC failed to offer the feature of allowing only trusted sources to write a variable. The rationale is that MAC follows the "no write down" principle to control write access, with which the information in a node can be written to another node if the security level of the former is the same or lower than the latter.

The model in [Samarati 1997] uses access control lists (ACLs) of objects to compute ACLs of executions (which are composed of one or more methods). A message filter is used to filter out possibly non-secure information flows. Interactions among executions are categorized into five modes. Different modes result in different security policies, which loosens the restriction of MAC. More flexibility is added by allowing exceptions during or after method execution [Ferrari 1997]. Relationships between the features mentioned in section 1 and the model in [Samarati 1997] are described below. First, it cannot detail the control granularity to different levels in different cases. In fact, it details the control granularity to objects only because ACLs are established among objects. Second, the model cannot adapt to dynamic object state change. The rationale is that ACLs are established among existing objects and therefore ACLs cannot be changed according to newly added objects during runtime. Third, purpose-oriented method invocation cannot be achieved because the control granularity details to objects only. Fourth, controlling method invocation through argument sensitivity and allowing only trusted sources to write a variable were not considered.

The purpose-oriented model [Yasuda 1997] proposes that invoking a method may be allowed for some methods but disallowed for others, even when the invokers belong to the same object. Relationships between the features mentioned in section 1 and the purpose-oriented model are described below. First, it cannot detail the control granularity to different levels in different cases. In fact, it details the control granularity to methods only. Second, the model cannot adapt to dynamic object state change. The rationale is that the model uses existing objects to create a flow graph, from which non-secure information flows can be identified. The flow graph is thus fixed during program execution. Third, purpose-oriented method invocation can be achieved. Fourth, controlling method invocation through argument sensitivity and allowing only trusted sources to write a variable were not considered.

The decentralized label approach [Myers 1998] [Myers 2000] marks the security levels of variables using labels. A label is composed of one or more policies, which should be simultaneously obeyed. A policy in a label is composed of an owner and zero or more readers that are allowed to read the data. Both owners and readers are principals, which may be users, group of users, and so on. Principals are grouped into hierarchies using the act-for relationships. A principal possesses all access rights of the principals it acts for. Join operation is used to avoid Trojan horses. Write access is controlled [Myers

2000]. Relationships between the features mentioned in section 1 and the decentralized label model are described below. First, it cannot detail the control granularity to different levels in different cases. In fact, it details the control granularity to variables because labels are attached to variables. Second, the model cannot adapt to dynamic object state change. The rationale is that, although principle hierarchies can be dynamically changed, principals seem fixed during runtime, which causes trouble when new objects are instantiated. Third, purpose-oriented method invocation was not considered. Fourth, controlling method invocation through argument sensitivity can be achieved but the author did not mention this. Fifth, the model controls write access more precise than other models [Myers 2000].

The approach in [McIlroy 1992] proposed a labeling system in UNIX. Every file, device, pipe, and process is attached with a label. Join operation is used to avoid Trojan horses. The approach also provides ceilings, which disallows processes to get into too sensitive locations. This avoids possible information leakage by the processes. The approach controls information flows among files, devices, and pipes. As to those among program variables, it does not control. Relationships between the features mentioned in section 1 and the model in [McIlroy 1992] are described below. First, it cannot detail the control granularity to different levels in different cases. In fact, it details the control granularity to objects only. Second, the model does not offer the features of adapting to dynamic object state change, purpose-oriented method invocation, controlling method invocation through argument sensitivity, and allowing only trusted sources to write a variable.

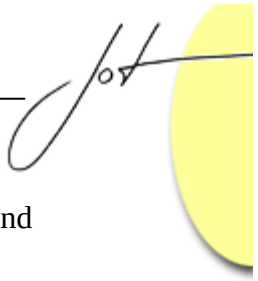
RBAC can also be used to control information flows. RBAC defines the roles a user can play. Users playing a role are generally human beings or intelligent agents [Sandhu 1996a]. A role is a collection of permissions [Sandhu 1996b]. When a user instantiates a session and plays a role in the session, the user possesses the permissions of the role. Permissions are revoked from the user when the user does not play the role or the session ends. A user can play multiple roles [Sandhu 1996a] and even change role during a session [Sandhu 1996b]. Inheritance and other relationships can be established among roles to structure them [Tari 1997] [Sandhu 1996b]. Moreover, constraints, such as two specific roles should be mutually exclusive, can be attached to roles [Ferraiolo 2001] [Giuri 1996] [Nyanchama 1999]. Relationships between the features mentioned in section 1 and RBAC are described below. First, it cannot detail the control granularity to different levels in different cases. In fact, it details the control granularity to only one level. Second, the model cannot adapt to dynamic object state change. The rationale is that users are the subjects that create sessions [Sandhu 1996b] and therefore users and sessions cannot be automatically managed by an application (i.e., the management of users and sessions cannot be programmed). Third, purpose-oriented method invocation can be achieved if permissions are defined as the legality of method invocation. Nevertheless, since the original design of RBAC regards users as human beings or agents [Sandhu 1996a], we cannot say that RBAC allows purpose orientation. Fourth, the features of controlling method invocation through argument sensitivity and allowing only trusted sources to write a variable are not offered.

The model in [Izaki 2001] uses RBAC to control information flows. It classifies object methods and derives a flow graph from method invocations. From the graph, non-secure information flows can be identified. Relationships between the features mentioned in section 1 and the model in [Izaki 2001] are described below. First, it cannot detail the control granularity to different levels in different cases. In fact, it details the control granularity to methods only. Second, the model cannot adapt to dynamic object state change because it uses predictable objects and methods to construct the flow graph. The flow graph thus cannot be changed during runtime. Third, purpose-oriented method invocation can be achieved because users in the model can be object methods. Fourth, the features of controlling method invocation through argument sensitivity and allowing only trusted sources to write a variable are not offered.

6 CONCLUSIONS

Role-based access control (RBAC) can be applied to control information flows (to protect privacy) within an application. The rationale is that RBAC is a super set of discretionary access control (DAC) and mandatory access control (MAC), which are useful in information flow control. Our research reveals that different control granularity is needed in different cases when controlling information flows. Currently we identify four levels of control granularity that should be offered simultaneously by an information flow control model, including the granularity that details to programs and files, that details to objects, that details to objects methods, and that details to variables. In the past years, we developed an RBAC-based information flow control model called OORBAC (object-oriented RBAC). It simultaneously details control granularity to the latter three levels. Nevertheless, the control mechanism in OORBAC is complicated. We thus revised OORBAC using the multiple-leveled RBAC approach, in which one level of RBAC offers one level of control granularity. We called the revised model L^{n}RBAC (n-leveled RBAC). Currently, L^{n}RBAC is composed of four levels. The first level (L0RBAC) controls the access rights from human beings or programs to programs or files. This level RBAC solves the covert channel problems induced by abnormal program stopping. The second level (L1RBAC) regulates the allowed and disallowed information flows among objects. The third level (L2RBAC) controls the legality of method invocations. And, the fourth level (L3RBAC) controls information flows among variables. L^{n}RBAC offers the following features:

1. It allows different control granularity in different cases
   This feature is a consequence of multiple-leveled RBAC, in which different level RBAC controls different granularity.
2. It solves the covert channel problems caused by abnormal program stopping
   This feature is achieved by L0RBAC. In that level, users are human beings or programs, and permissions are access rights from human beings or programs to



programs or files. With the permissions, unauthorized human beings and applications cannot access dumped files when a program is abnormally stopped.

3. It adapts to dynamic object state change
   This feature is achieved by L1RBAC and L2RBAC. When object state changes, L1RBAC decides whether information flows among objects are allowed and L2RBAC decides whether a method invocation is legal.
4. It controls method invocation through argument sensitivity
   This feature is achieved by L3RBAC, in which variables are independently assigned ACLs. With ACLs of variables, whether an argument is legal in an invocation can be checked by ACL comparison.
5. It allows purpose-oriented method invocation
   This feature is achieved by L2RBAC, which decides whether a method invocation is legal or not.
6. It controls write access precisely
   This feature is achieved by WACLs (write access control lists) and DSOURCEs (data sources) in L3RBAC.
7. It avoids Trojan horses
   This feature is achieved by the join operation.

REFERENCES

[Bell 1976] Bell D. E. and LaPadula, L. J., "Secure Computer Systems: Unified Exposition and Multics Interpretation," *Technique report, Mitre Corp.*, Mar. 1976. http://csrc.nist.gov/publications/history/bell76.pdf

[Chou in press] Chou, S. -C., "Embedding Role-Based Access Control Model in Object-Oriented Systems to Protect Privacy," to appear in *Journal of Systems and Software*.

[Denning 1976] Denning, D. E., "A Lattice Model of Secure Information Flow," *Comm. ACM*, vol. 19, no. 5, pp. 236-243, 1976.

[Denning 1977] Denning D. E. and Denning, P. J., "Certification of Program for Secure Information Flow," *Comm. ACM*, vol. 20, no. 7, pp. 504-513, 1977.

[Ferraiolo 2001] Ferraiolo, D. F., et al., "Proposed NIST Standard for Role-Based Access Control," *ACM Trans. Information and System Security*, vol. 4, no. 3, pp. 224-274, 2001.

[Ferrari 1997] Ferrari, E., et al., "Providing Flexibility in Information Flow Control for Object-Oriented Systems," *Proc. 13'th IEEE Symp. Security and Privacy*, pp. 130-140, 1997.

[Focardi 1997] Focardi R. and Gorrieri, R., "The Compositional Security Checker: A Tool for the Verification of Information Flow Security Properties," *IEEE Trans. Software Eng.*, vol. 23, no. 9, pp. 550-571, 1997.

[Giuri 1996] Giuri L. and Iglio, P., "A Formal Models for Role-Based Access Control with Constraints," *Proc. 9'th IEEE Computer Security Foundations Workshop*, pp. 136-145, 1996.

[Izaki 2001] Izaki, K., et al., "Information Flow Control in Role-Based Model for Distributed Objects," *Proc. 8'th International Conf. Parallel and Distributed Systems*, pp. 363-370, 2001.

[McCollum 1990] McCollum, C. J., et al., "Beyond the Pale of MAC and DAC - Defining New Forms of Access Control," *Proc. 6'th IEEE Symp. Security and Privacy*, pp. 190-200, 1990.

[McIlroy 1992] McIlroy M. D. and Reeds, J. A., "Multilevel Security in the UNIX Tradition," *Software - Practice and Experience*, vol. 22, no. 8, pp. 673-694, 1992.

[Myers 1998] Myers A. and Liskov, B., "Complete, Safe Information Flow with Decentralized Labels," *Proc. 14'th IEEE Symp. Security and Privacy*, pp. 186-197, 1998.

[Myers 2000] Myers A. and Liskov, B., "Protecting Privacy using the Decentralized Label Model," *ACM Trans. Software Eng. Methodology*, vol. 9, no. 4, pp. 410-442, 2000.

[Nyanchama 1999] Nyanchama M. and Osborn, S., "The Role Graph Model and Conflict of Interest," *ACM Tran. Info. Sys. Security*, vol. 2, no. 1, pp. 3-33, 1999.

[Osborn 2000] Osborn, S., et al., "Configuring Role-Based Access Control to Enforce Mandatory and Discretionary Access Control Policies," *ACM Trans. Info. Sys. Security*, vol. 3, no. 2, pp. 85-106, 2000.

[Pressman 2001] Pressman, R. S., *Software Engineering, A Practitioner's Approach, fifth edition*, pp. 305- 306, McGraw-Hill 2001.

[Rumbaugh 1999] Rumbaugh, J., et al., *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.

[Samarati 1997] Samarati, P., et al., "Information Flow Control in Object-Oriented Systems," *IEEE Trans. Knowledge Data Eng.*, vol. 9, no. 4, pp.524-538, Jul./Aug. 1997.

[Sandhu 1993] Sandhu, R. S., "Lattice-Based Access Control Models," *IEEE Computer*, vol. 26, no. 11, pp. 9-19, Nov. 1993.

[Sandhu 1996a] Sandhu, R., "Role Hierarchies and Constraints for Lattice-Based Access Controls," *Proc. Fourth European Symposium on Research in Computer Security*, pp. 65-79, 1996.

[Sandhu 1996b] Sandhu, R. S., et al., "Role-Based Access Control Models," *IEEE Computer*, vol. 29, no. 2, pp. 38-47, 1996.

[Tari 1997] Tari Z. and Chan, S. -W., "A Role-Based Access Control for Intranet Security," *IEEE Internet Computing*, vol. 1, no. 5, pp. 24-34, 1997.

[Varadharajan 1990] Varadharajan V. and Black, S., "A Multilevel Security Model for a Distributed Object-Oriented System," *Proc. 6'th IEEE Symp. Security and Privacy*, pp. 68-78, 1990.

[Yasuda 1997] Yasuda, M., et al., "Information Flow in a Purpose-Oriented Access Control Model," *Proc. 1997 International Conf. Parallel and Distributed Systems*, pp. 244-249, 1997.

About the author

Shih-Chien Chou received a Ph. D. degree from the Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan. He is currently an associate professor in the Department of Computer Science and Information Engineering, National Dong Hwa University, Hualien, Taiwan. His research interests include software engineering, process environment, software reuse, and information flow control. He can be contacted through the e-mail address scchou@mail.ndhu.edu.tw .

APPENDIX 1. The man/woman example mentioned in section 1. The object states in Figure 2 are used in the following implementation.

```
/*---------- Level 0 RBAC below ------------*/

1   L0RBAC {
    1.1 role operator{ // John plays this role
        1.1.1     a.exe(e), debugger.exe(e); // e: execution right
    1.2 }
    1.3 role program{ // a.exe plays this role
        1.3.1     file1.dat(r), file2.dat(w); // r: read right, w:
            write right
    1.4 }
    1.5 role debugger{ // debugger.exe plays this role
        1.5.1     dump.core(r);
    1.6 }
2   }

/*---------- Level 1 RBAC below ------------*/

3   L1RBAC {
    3.1 association friends {
        3.1.1     role man (0,*){
            3.1.1.1     woman (0,*);
        3.1.2     }
        3.1.3     role woman (0,*){
            3.1.3.1     man (0,*);
        3.1.4     }
        3.1.5     role example (1,M){
            3.1.5.1     // the class example can access every other
                class
        3.1.6     }
    3.2 }

    3.3 association married {
        3.3.1     role man (0,*){
            3.3.1.1     woman (0,*), certificate(M, 1);
        3.3.2     }
        3.3.3     role woman (0,*){
            3.3.3.1     man (0,*), certificate(M, 1);
        3.3.4     }
        3.3.5     role example (1,M){
            3.3.5.1     // the class example can access every other
                class
        3.3.6     }
```

```
   3.4 }
4  }

/*---------- Level 2 RBAC below ------------*/

5  L2RBAC {
   5.1 association friends {
      5.1.1     role man{
         5.1.1.1     get_others_general_info:
               woman.get_self_general_info;
      5.1.2      }
      5.1.3     role woman{
         5.1.3.1     get_others_general_info:
               man.get_self_general_info;
      5.1.4      }
      5.1.5     role example{
         5.1.5.1     // example.main can invoke every methods in
               other classes
      5.1.6      }
   5.2 }
   5.3 association married {
      5.3.1     rh: married≥friends // all the permissions in
          the association "friends" are inherited by the
          association "married"
      5.3.2     role man{
         5.3.2.1     get_others_personal_info:
               woman.get_self_personal_info;
         5.3.2.2     change_others_general_info:
               woman.change_self_general_info;
         5.3.2.3     get_certificate_info:
               certificate.get_certificate_info;
      5.3.3      }
      5.3.4     role woman{
         5.3.4.1     get_others_personal_info:
               man.get_self_personal_info;
         5.3.4.2     change_others_general_info:
               man.change_self_general_info;
         5.3.4.3     get_certificate_info:
               certificate.get_certificate_info;
      5.3.5      }
      5.3.6     role example{
         5.3.6.1     // example.main can invoke every methods in
               other classes
      5.3.7      }
   5.4 }
6  }
```

```
/*---------- Level 3 RBAC below ------------*/

7  L3RBAC {
   7.1 association friends {
      7.1.1     role man{
         7.1.1.1     self_general_info {get_self_general_info,
              woman.get_others_general_info; example.main};
         7.1.1.2     others_general_info
              {get_others_general_info; get_others_general_info,
              woman.get_self_general_info};
         7.1.1.3     get_self_general_info.return
              {woman.get_others_general_info; NONE}; // The
              return value of the method
              "man.get_self_general_info" can only be received
              by the method woman.get_others_general_info".
      7.1.2     }
      7.1.3     role woman{
         7.1.3.1     self_general_info {get_self_general_info,
              man.get_others_general_info; example.main};
         7.1.3.2     others_general_info
              {get_others_general_info; get_others_general_info,
              man.get_self_general_info};
         7.1.3.3     get_self_general_info.return
              {man.get_others_general_info; NONE}; // The return
              value of the method "woman.get_self_general_info"
              can only be received by the method
              man.get_others_general_info".
      7.1.4     }
   7.2 }

   7.3 association married {
      7.3.1     role man{
         7.3.1.1     self_general_info {get_self_general_info,
              woman.get_others_general_info,
              change_self_general_info;
              change_self_general_info,
              woman.change_others_general_info};
         7.3.1.2     others_general_info
              {get_others_general_info; get_others_general_info,
              woman.get_self_general_info};
         7.3.1.3     others_new_general_info
              {change_others_general_info,
              get_others_general_info,
              woman.change_self_general_info,
```

```
            woman.get_self_general_info;
            change_others_general_info};
         7.3.1.4     self_personal_info {get_self_personal_info,
            woman.get_others_personal_info; example.main};
         7.3.1.5     others_personal_info
            {get_others_personal_info;
            get_others_personal_info,
            woman.get_self_personal_info};
         7.3.1.6     get_self_general_info.return
            {woman.get_others_general_info; NONE};
         7.3.1.7     get_self_personal_info.return
            {woman.get_others_personal_info; NONE};
      7.3.2     }
      7.3.3     role woman{
         7.3.3.1     self_general_info {get_self_general_info,
            man.get_others_general_info,
            change_self_general_info;
            change_self_general_info,
            man.change_others_general_info};
         7.3.3.2     others_general_info
            {get_others_general_info; get_others_general_info,
            man.get_self_general_info};
         7.3.3.3     others_new_general_info
            {change_others_general_info,
            get_others_general_info,
            man.change_self_general_info,
            man.get_self_general_info;
            change_others_general_info};
         7.3.3.4     self_personal_info {get_self_personal_info,
            man.get_others_personal_info; example.main};
         7.3.3.5     others_personal_info
            {get_others_personal_info;
            get_others_personal_info,
            man.get_self_personal_info};
         7.3.3.6     get_self_general_info.return
            {man.get_others_general_info; NONE};
         7.3.3.7     get_self_personal_info.return
            {man.get_others_personal_info; NONE};
      7.3.4     }
      7.3.5     role certificate{
         7.3.5.1     certificate_info {man.get_certificate_info,
            woman.get_certificate_infor; example.main};
         7.3.5.2     get_certificate_info.return
            {{man.get_certificate_info,
            woman.get_certificate_infor; NONE};
      7.3.6     }
   7.4 }
```

```
8  }

/* --------- JAVA program below ---------------------*/

9  class man {
   9.1 public String self_personal_info, self_general_info,
       others_personal_info, others_general_info,
       others_new_general_info, certificate_info;
   9.2 public void get_others_general_info(woman w1){ // The
       method gets a woman's general information
      9.2.1     others_general_info = w1.get_self_general_info();
   9.3 }
   9.4 public void get_others_personal_info(woman w1){ // The
       method gets a woman's personal information
      9.4.1     others_personal_info =
          w1.get_self_personal_info();
   9.5 }
   9.6 public void get_certificate_info(certificate c1){ // The
       method gets the information of the man's marriage
       certification
      9.6.1     certificate_info = c1. get_certificate_info();
   9.7 }
   9.8 public void change_others_general_info(woman w1){ // The
       method changes a woman's general information
      9.8.1     /* set up new general information to
          others_new_general_info for the change */
      9.8.2     w1.change_self_general_info(others_new_general_in
          fo);
   9.9 }
   9.10public void change_self_general_info(String
       new_general_info){ // The method changes the general
       information of the man himself. It will be invoked by the
       method "change_others_general_info" of the woman class
      9.10.1    self_general_info = new_general_info;
   9.11}
   9.12public String get_sef_general_info(){ // The method gets
       the general information of the man himself. It will be
       invoked by the method "get_others_general_info" of the
       woman class.
      9.12.1    return self_general_info;
   9.13}
   9.14public String get_sef_personal_info(){ // The method gets
       the personal information of the man himself. It will be
       invoked by the method "get_others_personal_info" of the
       woman class.
      9.14.1    return self_personal_info;
```

```
    9.15}
10 } /* end of class "man" */

11 class woman {
    11.1public String self_personal_info, self_general_info,
      others_personal_info, others_general_info,
      others_new_general_info, certificate_info;
    11.2public void get_others_general_info(man m1){ // The
      method gets a man's general information
       11.2.1    others_general_info = m1.get_self_general_info();
    11.3}
    11.4public void get_others_personal_info(man m1){ // The
      method gets a man's personal information
       11.4.1    others_personal_info =
          m1.get_self_personal_info();
    11.5}
    11.6public void get_certificate_info(certificate c1){ // The
      method gets the information of the woman's marriage
      certification
       11.6.1    certificate_info = c1. get_certificate_info();
    11.7}
    11.8public void change_others_general_info(man m1){ // The
      method changes a man's general information
       11.8.1    /* set up new general information to
          others_new_general_info for the change */
       11.8.2    m1.change_self_general_info(others_new_general_in
          fo);
    11.9}
    11.10   public void change_self_general_info(String
      new_general_info){ // The method changes the general
      information of the woman herself. It will be invoked by
      the method "change_others_general_info" of the man class
       11.10.1   self_general_info = new_general_info;
    11.11   }
    11.12   public String get_sef_general_info(){ // The method
      gets the general information of the woman herself. It
      will be invoked by the method "get_others_general_info"
      of the man class.
       11.12.1   return self_general_info;
    11.13   }
    11.14   public String get_sef_personal_info(){ // The method
      gets the personal information of the woman herself. It
      will be invoked by the method "get_others_personal_info"
      of the man class.
       11.14.1   return self_personal_info;
    11.15   }
12 } /* end of class "woman" */
```

```
13 class certificate {
   13.1 public String certificate_info;
   13.2 public String get_certificate_info(){
      13.2.1    return certificate_info;
   13.3 }
14 }

15 class example {
   15.1 public man mm[]; // reserve an array for man
   15.2 public woman ww[]; // reserve an array for woman
   15.3 public certificate cer[]; // reserve an array for
        certificate
   15.4 public void main() {

// The following statements create the object state in Figure
2(a)
      15.4.1    // instantiate two men "m1" and "m2", and
          respectively assign them to the variables "mm[1]" and
          "mm[2]"
      15.4.2    // instantiate three women "w1", "w2", and "w3",
          and respectively assign them to the variables "ww[1]",
          "ww[2]", and ww[3]
      15.4.3    // instantiate a certificate "cer1" and assign it
          to the variable "cer[1]"
      15.4.4    addAG(association friends, man mm[2], woman
          ww[2]);
      15.4.5    addAG(association friends, man mm[2], woman
          ww[3]);
      15.4.6    addAG(association friends, man mm[1], woman
          ww[2]);
      15.4.7    addAG(association married, man mm[1], woman
          ww[1], certificate cer[1]);

// The following statements are allowed
      15.4.8    mm[1].get_others_personal_info(ww[1]);
      15.4.9    ww[1].change_others_general_info(mm[1]);
      15.4.10   ww[1].get_certificate_info(cer[1]);
      15.4.11   mm[2].get_others_general_info(ww[3]);
      15.4.12   mm[2].get_others_general_info(ww[2]);
      15.4.13   // . . .

// The following statements are disallowed
      15.4.14   mm[2].get_others_general_info(ww[1]); // mm[2]
          and ww[1] are strangers under the current object state
```

```
        15.4.15   mm[2].get_others_personal_info(ww[3]); // mm[2]
            and ww[3] are not married under the current object
            state
        15.4.16   mm[1].get_others_general_info(ww[3]); // mm[1]
            and ww[3] are strangers under the current object state
        15.4.17   mm[1].change_others_general_info(ww[2]); // mm[1]
            and ww[2] are not married under the current object
            state
        15.4.18   // . . .

// The following statements create the object state in Figure
2(b)
        15.4.19   // instantiate a woman "w4" and assign her to the
            variable "ww[4]"
        15.4.20   // instantiate three certificate "cer2", "cer3",
            and cer4", and respectively assign them to the
            variables "cer[2]", "cer[3]", "cer[4]"
        15.4.21   removeAG(association friends, man mm[2], woman
            ww[2]);
        15.4.22   removeAG(association friends, man mm[2], woman
            ww[3]);
        15.4.23   removeAG(association friends, man mm[1], woman
            ww[2]);
        15.4.24   removeAG(association married, man mm[1], woman
            ww[1], certificate cer[1]);
        15.4.25   // de-allocate cer[1] // if cer[1] is not de-
            allocated, the modality constraint will be violated
        15.4.26   addAG(association married, man mm[1], woman
            ww[2], certificate cer[2]);
        15.4.27   addAG(association married, man mm[2], woman
            ww[3], certificate cer[3]); // This statement is not
            allowed because it violate the cardinality constraint
        15.4.28   addAG(association married, man mm[2], woman
            ww[3], certificate cer[3]);
        15.4.29   addAG(association married, man mm[3], woman
            ww[4], certificate cer[4]);
        15.4.30   addAG(association friends, man mm[2], woman
            ww[4]);

// The following statements are allowed
        15.4.31   mm[2].get_others_personal_info(ww[3]); // mm[2]
            and ww[3] are married under the current object state.
            This statement in line 15.4.15 is illegal
        15.4.32   mm[1].change_others_general_info(ww[2]); // mm[1]
            and ww[2] are married under the current object state.
            This statement in line 15.4.17 is illegal
        15.4.33   ww[4].get_certificate_info(cer[4]);
```

```
      15.4.34   mm[2].get_others_personal_info(ww[3]);
      15.4.35   mm[2].change_others_general_info(ww[3]);
      15.4.36   mm[3].get_others_general_info(ww[3]);
      15.4.37   mm[2].get_certificate_info(cer[3]);
      15.4.38   // . . .

// The following statements are disallowed
      15.4.39   mm[1].get_others_personal_info(ww[1]); // this
          statement in line 15.4.8 is allowed
      15.4.40   ww[1].change_others_general_info(mm[1]); // this
          statement in line 15.4.9 is allowed
      15.4.41   ww[1].get_certificate_info(cer[1]); // this
          statement in line 15.4.10 is allowed
      15.4.42   mm[2].get_others_general_info(ww[2]); // this
          statement in line 15.4.12 is allowed

// The following statements create the object state in Figure
2(c)
      15.4.43   // de-allocate ww[3] // ww[3] passed away

// The following statements is still allowed in spite of ww[3]'s
passing away
      15.4.44   mm[2].get_certificate_info(cer[3]);

// The following statements are disallowed because the object
"ww[3]" is not existing
      15.4.45   mm[2].get_others_personal_info(ww[3]);
      15.4.46   mm[2].change_others_general_info(ww[3]);
      15.4.47   mm[3].get_others_general_info(ww[3]);
   15.5} // end of "main"
16 } // end of class "example"
```