I.J. Information Technology and Computer Science, **2013, 04, 1-13**
Published Online March 2013 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijitcs.2013.04.01

A Partial Ordered Number System for Information Flow Control

Shih-Chien Chou
Department of Computer Science and Information Engineering, National Dong Hwa University, Taiwan
scchou@mail.ndhu.edu.tw

Abstract— *Information flow control models* can be applied widely. This paper discusses only the models preventing information leakage during program execution. In the prevention, an information flow control model dynamically monitors statements that will cause information flows and ban statements that may cause leakage. We involved in the research of information flow control for years and identified that sensitive information may be leaked only when it is output. However, most existing models ignore information flows induced by output statements. We thus designed a new model *XIFC* (X information flow control) that especially emphasizes the monitoring of output statements. We also designed XIFC as a precise and low runtime overhead model. To achieve this purpose, we took a different viewpoint to re-examine the features offered by existing models and extracted a necessary feature set for the design. Our experiments show that XIFC bans every non-secure information flow and substantially reduces runtime overhead when comparing with our previous work.

Index Terms— Information Flow Control, Information Leakage Prevention, Security, Access Control, Partial Ordered

I. Introduction

Information flow control models can ensure secure database interfaces [1], ensure secure information flows within an operating system [2][3] and among distributed operating systems [4], prevent information leakage during program execution, prevent information leakage within web services [5][6], and ensure the security of information flowing forward to and backward from cascading web services [7][8]. Perhaps preventing information leakage during program execution is the earliest application of information flow control. This paper discusses only the control and excludes others. Therefore, an information flow control model mentioned in the rest of this paper is a model that prevents information leakage during program execution.

To achieve the prevention, an information flow control model monitors statements that will cause information flows and ban statements that may leak information. For example, if high sensitive information is required to output to a low sensitive device, the output may be non-secure and therefore should be banned (here “sensitive information” is the information that should be protected). Information flows may occur when: (a) assigning an expression result to a variable such as “a=b+c;”, (b) invoking a component and returning from an invocation, (c) reading input media such as “scanf(“%d”, &x);”, (d) writing output media such as “printf(“%d”, x);”, and (e) sending information to another program. Since an information flow control model prevents information leakage, we have to explain the term “information leakage”. Generally, *information leakage occurs when sensitive information managed by a software system is illegally obtained by persons or other software*. Our research excludes malicious software such as viruses and worms. Therefore, *sensitive information may be leaked only when it is output* because persons and non-malicious software can only access output information (note that sending information to other software is a special type of output). According to our survey, *most existing models monitor information flows within a system but ignore the flows induced by output statements*. Accordingly, our new design especially emphasizes the monitoring of information output.

We involved in the research of information flow control for years and identified that precise control and low runtime overhead are difficult to achieve simultaneously. That is, a comprehensive model can ban every non-secure information flow but may induce high runtime overhead. On the other hand, a low runtime overhead model may be imprecise. To design a precise and low runtime overhead model, we re-examined the features offered by existing models and extracted a necessary feature set from a different viewpoint. The extracted features is used to design a new model, which bans the minimum set of information flows but still ensured no information leakage. After a long time of trial and error, we designed a model using a partial ordered number system. It is named *XIFC* (X information flow control) in which the letter “X” indicates that the concepts in XIFC deviate from existing models. Primary features offered by XIFC are listed below, in which the latter four features are all useful in reducing runtime overhead.

a. XIFC prevents information leakage during program execution. This feature should be offered by every information flow control model.

b. XIFC uses simple partial ordered numbers instead of complicated mechanisms such as decentralized labels and role-based permissions to represent security levels. Checking the validity of information flows using number comparison is expected to be faster than using the complicated mechanisms.

c. XIFC strictly monitors every output statement but allows most other ones because only output may leak information. To allow most non-output statements, the join operator is applied to ensure no information leakage will occur.

d. XIFC only monitors information flows involving sensitive variables and/or sensitive I/O media. The runtime overhead will be smaller as the number of sensitive information is less.

e. XIFC uses bits to represent security levels. With this, bit operations can be applied to reduce runtime overhead.

Although we just mention five features, XIFC offers more important ones described in section 3.2. According to our survey, most existing models only offer the about mentioned first feature but not the others, especially the third one. In the rest of this paper, we discuss the extraction of features and the design of XIFC. Our experiments show that XIFC bans every non-secure information flow and reduces runtime overhead substantially when comparing with our previous work. In the rest of the paper, section II discusses related work, section III describes feature extraction, section IV describes XIFC, section V shows an example, section VI proves the correctness of XIFC, section VII gives the evaluation of XIFC, and section VIII is the conclusion.

II. Related Work

Access control matrix (ACM) [9] allows a subject to access an object if the subject possesses the access right. ACM generally achieves static but not dynamic access control [10][11], such as changing rights using the join operator [12]. DACM (dynamic ACM) [10] allows dynamically granting access rights under different situations. ACM and its variants such as capability lists are discretionary access control (DAC) [13].

Mandatory access control (MAC) [14][15][16][17] is also useful in access control. The MAC model proposed by Bell&LaPadula [14] categorizes the security levels of objects. Subjects and information flows follow the "no read up" and "no write down" rules [14]. The lattice model [15][16] is a generalization of Bell&LaPadula's model (see [18] for a survey of lattice models). A lattice is defined as (SC, $\rightarrow$, $\oplus$), in which "SC" is the set of security classes, "$\rightarrow$" is the "can flow" relationship, and "$\oplus$" is the join operator. The "can flow" relationship controls information flows and the join operator prevent indirect information leakage [12].

The model in [19] is based on DAC, which controls information flows within object-oriented systems. ACLs (access control lists) of objects are used to compute ACLs of executions which are composed of object method(s). Possibly non-secure information flows are filtered out by a message filter. Interactions among executions are categorized into five modes to apply different security policies. Flexibility is added by allowing exceptions during or after method execution [20][21]. More flexibility is added using versions [22].

The purpose-oriented model [23][24][25] proposes that invoking a method may be allowed for some methods but disallowed for others, even when the invokers belong to the same object. Since different methods may be in different security levels [26], the consideration of purpose orientation is correct.

The approach in [27] proposed a labeling system for UNIX. It attaches a label to every file, device, pipe, and process. It controls information flows among files, devices, and pipes but not the information flows among program variables. It is thus considered insufficient in controlling information flows within a program.

The decentralized label model [28][29][30] attaches labels to variables. The security levels of variables are shown in the labels. A label is composed of one or more policies that should be simultaneously obeyed. In general, a policy is composed of an owner and zero or more readers that can read the data. Both owners and readers are principals, which may be users, group of users, and so on.

RBAC [31][32][33][34][35][36][37] can also be used in access control. It is composed of users, roles, sessions, permissions, role hierarchies, user-role assignments (URA), role-permission assignments (RPA), and constraints. A role is composed of a set of permissions [33], which is a consequence of RPA. Roles are structured using the "$\geq$" relationship. If a relationship "x $\geq$ y" exists, "x" possesses all the permissions of "y". The "$\geq$" relationship can thus be used to construct role hierarchies. Roles are assigned to users, which result in URA. Users can establish sessions, within which a user possesses the permissions of the role assigned to him. RBAC has been proved to be a super set of DAC and MAC [33][34][35][36][37]. Since DAC and MAC are useful in information flow control [14][15][16][17][19][20][21][22], RBAC can also be applied in the control. The model in [38] applies RBAC to control information flows in object-oriented systems. It classifies methods and derives a flow graph from which non-secure information flows can be identified. We also developed RBAC-based model [39]. It offers a read access rule to prevent information leakage and a write access rule to prevent information corruption.

The model in [40] uses read access control lists (RACL), write access control lists (WACL), permission lists (PERL), and create access lists (CACL) to determine whether an object method can invoke another one. The rules to determine the validity of an invocation are: (a) the object containing the invoking method has permissions to invoke the invoked method and (b) the object containing the invoked method has permission to access the parameters sent to the invoked method.

Flume [2] is a decentralized information flow control (DIFC) model for operating systems. It tracks information flows in a system using tags and labels. The control granularity is detailed to processes (i.e., Flume regards the information input to and output from a process as a whole). The secrecy tags prevent information leakage and the integrity tags prevent information corruption. The two types of controls are similar to the read and write control in our discussion. Flume also avoids information leaked to untrusted channel (e.g., sockets). The function of Laminar [3] is similar to that of Flume. Nevertheless, the control granularity is detailed to data structures (e.g., arrays) and system resources (e.g., sockets). Both Flume and Laminar are used in operating systems. Since our research focuses on embedding an information flow control model within a program to prevent information leakage, other models including Flume and Laminar are excluded in this paper.

III. Feature Extraction

In this section, we list the features offered by existing models and extract a set of necessary ones from our viewpoint to design XIFC.

3.1 Important Features Offered by Existing Models

Below we discuss the important features of information flow control we collected.

a. *Security levels of variables* ("security level" represent the sensitivity of a variable). To control information flows, sensitive variables should be associated with security levels such as ACLs and permissions. This is the basic feature offered by every existing model.

b. *Read access control* (*secrecy control in* [2][3]). Low security level variables cannot receive high sensitive information. This is the basic feature offered by every existing model.

c. *Partial ordering of security levels*. If numbers are used to represent security levels, the numbers may be partial ordered. This is an important feature offered by the lattice model [15][16].

d. *Use of labels*. A variable is associated with a label composed of one or more policies that should be simultaneously obeyed. This is the kernel feature offered by the decentralized label model.

e. *RBAC-embedding*. RBAC is embedded in a software system. Components of RBAC are used to control the execution of the software. This is the kernel feature offered by the RBAC-based models.

f. *Role embedding*. Some models do not embed the entire RBAC model but embed the concepts of roles and role hierarchies.

g. *Use of ACLs*. ACLs are applied to control object access. This is the kernel feature offered by the model in [19].

h. *Join operation*. After a variable *var* receives the information derived from a set of variables, the join operation is applied to adjust the security level of *var*. The operation prevents both direct and indirect information leakage. It is an important feature offered by almost every existing model.

i. *Dynamically changing security levels of variables*. The security levels of variables may be changed according to assignments. Most existing models allow the change using operators such as join.

j. *Declassification*. If a low sensitive variable possesses a high security level, the security level should be declassified. For example, the case history of a patient is sensitive and should be protected. However, the statistic information of ten thousands patients' case histories becomes non-sensitive because extracting the case history of a specific patient from the information is impossible. In this case, the security level of the statistic information should be declassified (the security level of the statistic information will be high because of the join operations).

k. *Granularity of control*. Control granularity of different models may be different. For example, the control granularity of the model in [19], that in [23][24][25][38][40], and that in [39] are respectively detailed to objects, methods, and variables.

l. *Applicable systems*. Some models are designed for object-oriented systems [19][39][40], some are for non-object-oriented systems [41], and some are for both types of systems [15][16][30].

m. *Static analysis*. The model in [38] statically constructs a method invocation graph and identifies non-secure information flows from the graph.

n. *Purpose-oriented invocation*. The model in [23][24][25] proposes that invoking a method may be allowed for some methods but disallowed for others according to different purposes.

o. *Role relationship management*. Our previous work [42] proposed that different role relationships may result in different permissions. For example, the discount rate given to a customer may be different for different relationships (e.g., strangers, friends, and family members will receive different discount rates).

p. *Write access control* (*integrity control in* [2][3]). Our previous work [39] proposed that a variable can only receive the information from trusted data sources. This control prevents information corruption.

q. *Controlling every variable*. If the control granularity of a model is detailed to variables, the model may control the information flows of every variable.

r. *Banning every non-secure information flow*. Existing models generally ban every non-secure information flow, except one of our previous work (see below).

s. *Allowing non-secure but harmless information flows; banning secure but harmful ones*. Since information may be leaked only when it is output, our previous work allows non-output information flows to execute until output occurs [43]. The security of the output will then be checked.

t. *Separation-of-duty* (*SoD*). This is an important constraint of RBAC. It is naturally that an RBAC-based model offers this feature.

u. *Controlling inter-program information flows*. Some models control the information exchanged among different organizations [17][44]. This control is similar to controlling *inter-program* information flows.

3.2 Extracting Necessary Features

A set of simple but necessary features are extracted according to our viewpoint. Perhaps others may extract other sets according to their viewpoints. We do not argue which viewpoint is better. We only hope to use the extracted features to design a precise and low runtime overhead model. The extracted features are listed below.

a. *The control granularity is detailed to variables*. We need this granularity of control because different variables carry information with different sensitivities. We use an example to explain this. Suppose a doctor can read and change the case history of a patient assigned to him, and can read but not change the case histories of others. Also suppose the patient *pt1* is assigned to the doctor *dc2* but not *dc1*. This implies that *dc1* can read the case history of *pt1* by invoking method(s) of *pt1*, and *dc2* can read and change the case history of *pt1* by invoking other method(s) of *pt1*. If the control granularity is detailed to objects, access rights are defined among objects. With this, both *dc1* and *dc2* can invoke every method of *pt1*, which incorrectly allows *dc1* to change the case history of *pt1*. If the control granularity is detailed to methods and *dc1* can invoke the method *pt1.md1* to read the case history of *pt1*. If a statement that changes the case history incorrectly appears in *pt1.md1*, the case history of *pt1* will be changed by the unauthorized doctor *dc1* when *dc1* invokes *pt1.md1*.

b. The security levels of variables are depicted by numbers. Existing models use mechanisms such as ACLs and permissions to depict security levels. Monitoring information flows using the mechanisms is time consuming when comparing with number comparison. In our new design, we use bits to depict security levels. This further reduces monitor time because bit operations can be applied.

c. The security levels of variables are partial ordered. Variables may be incomparable. For example, variables storing member numbers and those storing salaries are incomparable. This induces the needs of partial ordering. Partial ordering bans information comparison/exchange among variables that are incomparable. We use *groups* to achieve partial ordering. Only variables in the same group are comparable.

d. The join operation is applied. When a variable receives information, its security level should be adjusted to be the same as the information. This prevents both direct and indirect information leakage.

e. The declassification operation is applied. As described in the example of section 3.1, this feature is important and necessary.

f. Role concept is not applied inside a program but applied outside it. An executing program is an information exchange center. As long as no information is output, no information leakage will occur. Therefore, role concepts need not be embed in a program. However, information will be output eventually and the information may be captured by unauthorized roles (played by users or other software). Therefore, role concept should be applied outside a program. Since a program cannot control information outside it, the role concept is actually managed by the operating system. This implies that the information flow control model should cooperate with the operating system. The cooperation occurs on the I/O media (devices or files). That is, both I/O media and roles should possess security levels. When a role intends to access the information of an I/O media, the operating system should check the security levels to ensure the security of access.

g. Control inter-program information flows. Sending information to other programs is a type of output. Since outputting information may induce information leakage, inter-program information flows should be controlled.

h. Control sensitive variables only. Most existing models control information flows of every variable. Our new design does not control the flows involving only non-sensitive information to reduce runtime overhead.

In addition to the features mentioned above, we need two new features. The first is "*every output operations* *should be secure*" because only output will cause leakage. The second is "*variables should be allowed to change group*". For example, if a variable stores a salary with the unit USD and another one stores a salary with the unit NTD, they should belong to different groups. If the unit in the former variable is changed to be NTD by applying the currency exchange rate, the group of the former variable should be changed to be the same as the latter. We explain the extraction in the following paragraphs.

We do not mention read/write access control, except information output, because only output may leak information. We use an assignment to explain our consideration because assignment can be regarded as both read and write. For example, the assignment "a=b+c;" can be regarded that *a* read the information obtained from "b+c". It can also be regarded that the information obtained from "b+c" is written to *a*. Suppose variables in the assignment are in the same group (otherwise the statement cannot be executed) and the initial security level numbers of *a*, *b*, and *c* are respectively *1*, *2*, and *3*. Since the security level of the variable *a* is the lowest, existing models will ban the statement. However, as the join operation will change the security level number of *a* to be 3 after the assignment, the assignment will not cause information leakage. This explains why we do not control read access. To explain write access, we suppose the initial security numbers of *a*, *b*, and *c* are respectively *4*, *2*, and *3*. Since the security level of variable *a* is the highest this time, existing model allows the assignment. However, we think that the data sources of *b* and *c* should be trusted by *a* to prevent information corruption (this is "write access control" [39]). In our new design, we use groups to control write access. That is, if the information obtained from "b+c" cannot be written to *a*, we place the variables in different groups.

The doctor/patient example mentioned near the beginning of section 3.2 reveals the necessity of differentiating whether an assignment is a read or a write sometimes. Therefore, a variable should be associated with a read group and a write group. In the doctor/patient example, suppose the patient *pt1* is not assigned to the doctor *dc1*. Then, the assignment "dc1.pt1CaseHistory = pt1.caseHistory;" can be executed but not "pt1.caseHistory = dc1.pt1CaseHistory;" (*dc1* reads *pt1*'s case history in the former statement and write it in the latter one). To achieve this control, the variables *pt1.caseHistory* and *dc1.pt1CaseHistory* should be in the same read group but in the different write groups. Moreover, a mechanism indicating that the former assignment is a read and the latter a write should be available.

The spirit of the features "*use of labels*", "*RBAC-embedding*", "*role embedding*", and "*use of ACLs*" is similar. They assign permissions to roles (or principal). Our model [39] showed that using permissions induces large runtime overhead. We thus ignore permissions/roles inside a program but use partial ordered numbers to represent security levels of variables. However, the fact that users play roles cannot be vetoed in a software system. We thus incorporate the role concept outside the software. That is, the operating system should manage users and roles.

The feature "*dynamically change security levels of variables*" is achieved by the join operation. As to the feature "*applicable systems*", we think that object orientation or other paradigms will not affect the function of a program. Therefore, an information flow control model should be applicable to software of every paradigm. The feature "*static analysis*" is difficult to achieve because security levels will be changed dynamically. The features "*purpose-oriented invocation*" can be achieved easily when the control granularity is detailed to variables. For example, suppose the method "a.withdraw" can be invoked by "b.hkeeping" but not "b.drnk". Also suppose that: (a) the return value of "a.withdraw" is stored in the variable "a.mny" and (b) the variables "b.hk" and "b.dr" respectively receive the return values for "b.hkeeping" and "b.drnk". The purpose-oriented invocation can be achieved by placing "a.mny" and "b.hk" in the same group but placing "a.mny" and "b.dr" in different ones.

To achieve the feature "*role relationship management*", use different groups to store variables of different relationships. The feature "*control every variable*" is replaced by "*control sensitive variables only*", because non-sensitive information flows need not be controlled. The features "*ban every non-secure information flow*" and "*allow non-secure but harmless information flows; ban secure but harmful ones*" is replaced by the new feature "*every output operations should be secure*". The feature "*separation-of-duty (SoD)*" is related to role-based access control. Since roles are managed by the operating system, SoD should also be managed by it.

IV. XIFC

This section defines XIFC and describes the use of the model.

4.1 Definitions

XIFC uses a partial ordered number system to control information flows. A partial ordered number in XIFC is called a *security level* (*SL*), which depicts the sensitivities of variables and *I/O media* (devices and files). A *SL* is defined below.

Definition 1. *SL* = (*Grw*, *Gr*, *Gw*, *SLV*), in which

a. *Grw* = {*grw* | *grw* is a group number to control both read and write access}.

b. *Gr* = {*gr* | *gr* is a group number to control read access}.

c. *Gw* = {*gw* | *gw* is a group number to control write access}.

d. *SLV* is the security level number. Larger number corresponds to more sensitive information.

Some may think that *Grw* is redundant because of *Gr* and *Gw*. We introduce the redundancy to reduce runtime overhead. For a read access, only *Gr* is checked. For a write access, only *Gw* is checked. For assignment statements not differentiated as a read or a write, only *Grw* is checked (i.e., no need to check both *Gr* and *Gw*). XIFC only attaches *SL*s to sensitive variables (i.e., variables storing sensitive information) and sensitive I/O media. In Definition 1, a group in a *SL* may contain multiple numbers because a variable may be used in multiple situations. For example, a customer's member number may decide the discount rates and the airplane classes.

In most cases, XIFC uses a bit to represent a group number. For example, suppose a 32 bit word is used to represent a group. Then, the group 10010…100 indicates that the group numbers constitute the set {2, 28, 31}. If a bit represent a group number, checking whether variables are comparable can be achieved using bit ANDs (a non-zero AND result means group comparable). However, using bits to represent group numbers may cause trouble in some systems. For example, the case histories of different patients should be in different groups because a patient's case history can only be accessed by the patient himself. In this case, numbers instead of bits should be used for groups. In addition to groups, XIFC also uses bits to represent *SLV*s in which only one bit in a *SLV* is 1. For example, the *SLV*s 1000…00 and 000…001 in a 32 bit word are respectively 31 and 0. When using bits to represent a *SLV*, the largest *SLV* in a variable set can be identified using bit ORs followed by assembly instructions to extract the most significant bit from the bit OR result.

*SL*s control information flows within a program. If information should be sent to other programs, the information should be associated with a set of valid destinations (*VD*). A *VD* contains the programs that can receive the information, as defined below.

Definition 2. *VD* = {(*IPAdd*, *PortNum*)}, in which

a. *IPAdd* is the IP address of the site a program located.

b. *PortNum* is the port number assigned to the program.

*SL*s and *VD*s are associated with sensitive variables. According to Definition 2, a *VD* is a set, which means that more than one program may be allowed to receive the information of a variable. As to non-sensitive variables, they have no *SL*s and *VD*s. When checking *SL*s and *VD*s, non-sensitive variables are bypassed. After defining *SL* and *VD*, XIFC is defined below.

Definition 3. *XIFC* = (*SVAR*, *SIO*, *SLS*, *VDS*, *JOIN*, *DECL*, *CTLM*), in which

a. *SVAR* is the set of sensitive variables. Every sensitive variable is associated with a *SL* and a *VD*.

b. *SIO* is the set of sensitive I/O media. Every sensitive I/O media is associated with a *SL* but not *VD* because a media cannot be sent to a program. The *SL*s of I/O media are defined according to the media's location. The only possibility to change an I/O media's *SL* is changing its location. Since a program does not know the change, changing the *SL* is out of the scope of XIFC. This means that the *SL* of I/O media will keep unchanged during program execution. As to files (which are also I/O media), their *SL*s also cannot be changed. For example, if the *SLV* (see Definition 1) of a file is *n*, it can be accessed by roles possessing a privilege to access files whose *SLV* is *n* or smaller. If the *SLV* is increased, the roles can no longer access the file.

c. *SLS* is the set of *SL*s associated with sensitive variables and I/O media.

d. *VDS* is the set of *VD*s associated with sensitive variables.

e. *JOIN* is the join operator. It will be described in more details later.

f. *DECL* is the declassification operator.

g. *CTLM* is the information flow control mechanisms embedded in a program. XIFC use directives for the control.

4.2 Using XIFC

We use the five types of statements that will result in information flows mentioned near the beginning of section 1 to describe the use of XIFC.

a. Assignment statements

To control information flows, only statements that may cause information leakage or corruption should be monitored. Since only output statements may cause leakage, the monitoring of assignments focuses on preventing information corruption. The prevention can be achieved by checking whether variables are comparable (i.e., checking whether variables are within the same group).

For an assignment without sensitive variables, the statement can be executed and XIFC does nothing. If a sensitive variable *var* is assigned a value derived from non-sensitive ones, the assignment is allowed. After that, *var* becomes non-sensitive.

For an assignment categorized as a read access, suppose: (a) *Rvar* reads the value derived from the sensitive variables in the set $\{var_i\}$ and other non-sensitive ones, (b) the *SL* and *VD* of var_i are respectively SL_i and VD_i, (c) $SL_i = (Grw_i, Gr_i, Gw_i, SLV_i)$, and (d) the *SL* of *Rvar* is $(Grw_{Rvar}, Gr_{Rvar}, Gw_{Rvar}, SLV_{Rvar})$ and the *VD* of *Rvar* is VD_{Rvar}. *Note that the SL and VD of a non-sensitive variable are composed of blank fields.* The assignment can be executed if the set

" $(\cap_i Gr_i) \cap Gr_{Rvar}$" is not empty (if blank fields appear in the intersection, ignore it). The intersections can be achieved by bit ANDs if bits represent groups. If numbers represents groups, assembly instructions are used. After the assignment, the join operator performs the following operations.

$$\begin{aligned} Gr_{Rvar} &= \cap_i Gr_i \\ Grw_{Rvar} &= Gr_{Rvar} \cap Gw_{Rvar} \\ SLV_{Rvar} &= MAX(SLV_i) \\ VD_{Rvar} &= \cap_i VD_i \end{aligned} \tag{J1}$$

In the operations, *MAX* retrieves the maximum one from a set of values. The join operations are implemented using assembly instructions to reduce runtime overhead. According to the join, a non-sensitive variable receiving sensitive information will become sensitive to prevent leakage.

For an assignment categorized as a write access, suppose: (a) *Wvar* is written by the value derived from the sensitive variables in the set $\{var_i\}$ and other non-sensitive ones, (b) the *SL* and *VD* of var_i are respectively SL_i and VD_i, (c) SL_i = (Grw_i, Gr_i, Gw_i, SLV_i), and (d) the *SL* of *Wvar* is (Grw_{Wvar}, Gr_{Wvar}, Gw_{Wvar}, SLV_{Wvar}) and the *VD* of *Wvar* is VD_{Wvar}. The assignment can be executed if the set " $(\cap_i Gw_i) \cap Gw_{Wvar}$" is not empty. After the assignment, the join operator performs the following operations.

$$\begin{aligned} Gw_{Wvar} &= \cap_i Gw_i \\ Grw_{Wvar} &= Gr_{Wvar} \cap Gw_{Wvar} \\ SLV_{Wvar} &= MAX(SLV_i) \\ VD_{Wvar} &= \cap_i VD_i \end{aligned} \tag{J2}$$

For an assignment not categorized as a read or a write, suppose: (a) *RWvar* is assigned the value derived from the sensitive variables in the set $\{var_i\}$ and other non-sensitive ones, (b) the *SL* and *VD* of var_i are respectively SL_i and VD_i, (c) SL_i = (Grw_i, Gr_i, Gw_i, SLV_i), and (d) the *SL* of *RWvar* is (Grw_{RWvar}, Gr_{RWvar}, Gw_{RWvar}, SLV_{RWvar}) and the *VD* of *RWvar* is VD_{RWvar}. The assignment can be executed if " $(\cap_i Grw_i) \cap Grw_{RWvar}$" is not empty. After the assignment, the join operator performs the following operations.

$$\begin{aligned} Gr_{RWvar} = Gw_{RWvar} = Grw_{RWvar} &= \cap_i Grw_i \\ SLV_{RWvar} &= MAX(SLV_i) \\ VD_{RWvar} &= \cap_i VD_i \end{aligned} \tag{J3}$$

b. Invocation and returning from an invocation

During an invocation, the arguments of an invoking component (e.g., a C function) are assigned to the parameters of an invoked one. The information flows induced by an invocation are thus similar to an assignment not categorized as a read or a write. Accordingly, the management for an assignment and the join operation set J3 can be applied for an invocation. As to returning from an invocation, the return information is assigned to a variable. This is again similar to an assignment. It can thus be handled similar to an invocation.

c. Statements that read information from input media

A read operation will not output information. Therefore, only read groups should be checked for variable comparability. If the checking passes, the read operation is allowed. A read operation may obtain information from input devices or files. Manipulating the two types of read operations is similar. If the variable *var* intends to read information from an input media *ime* (*ime* may be an input device or a file), the read operation is decided as follows.

c.1 If both *var* and *ime* are non-sensitive, the read operation is allowed and XIFC does nothing.

c.2 If *var* is sensitive but *ime* is non-sensitive, the read operation is allowed. After that, *var* becomes non-sensitive.

c.3 If *var* is non-sensitive but *ime* is sensitive, the read operation is allowed.

c.4 If both *var* and *ime* are sensitive, the read operation is allowed if the intersection of their read group is not empty.

After reading an input device, the join operator performs the operations below (Gw_{var} and VD_{var} are unchanged because an input device has no *Gw* and *VD*).

$$\begin{aligned} Gr_{var} &= Gr_{ime} \\ Grw_{var} &= Gr_{var} \cap Gw_{var} \\ SLV_{var} &= SLV_{ime} \end{aligned} \tag{J4}$$

After reading a file, the join operator performs the operations below (here we suppose the variable *var* reads the information *inf* from the file *ime*).

$$\begin{aligned} Gr_{var} &= Gr_{inf} \\ Gw_{var} &= Gw_{inf} \\ Grw_{var} &= Gr_{var} \cap Gw_{var} \\ SLV_{var} &= SLV_{inf} \\ VD_{var} &= VD_{inf} \end{aligned} \tag{J5}$$

d. Statements that write information to output media

Only information output may cause leakage. Therefore, XIFC controls output strictly. To achieve the control, every sensitive output device and file is associated with a *SL*. If the information *inf* is required to output to the media (devices or files) *odev*, the output operation is decided as follows.

Only information output may cause leakage. Therefore, XIFC controls output strictly. To achieve the control, every sensitive output device and file is associated with a *SL*. If the information *inf* is required to output to the media (devices or files) *odev*, the output operation is decided as follows.

d.1 If *inf* is non-sensitive, the output is allowed and XIFC does nothing.

d.2 If *inf* is sensitive but *odev* is non-sensitive, the output is banned.

d.3 If both *inf* and *odev* are sensitive, the output should be checked. Suppose: (a) the *SL*s of *inf* and *odev* are respectively SL_{inf} and SL_{odev}, (b) $SL_{inf} = (Grw_{inf}, Gr_{inf}, Gw_{inf}, SLV_{inf})$, and (c) $SL_{odev} = (Grw_{odev}, Gr_{odev}, Gw_{odev}, SLV_{odev})$. Then, the output is allowed only when the condition "$(Gw_{\inf} \cap Gw_{odev} \neq \phi) \wedge (SLV_{odev} \geq SLV_{\inf})$" is true.

When outputting information to a file, the *SL* and *VD* of the information should also be output to protect the information. No join operations are needed after the output because the *SL* of an output media cannot be changed.

When outputting information to a file, the *SL* and *VD* of the information should also be output to protect the information. No join operations are needed after the output because the *SL* of an output media cannot be changed.

e. Statements that send information to another program

If program *prg1* intends to send information to *prg2*, *prg2* should be embedded with XIFC. Sending non-sensitive information is allowed and XIFC does nothing. To send the sensitive information *inf* to *prg2*, the IP address and port number pair of *prg2* should be within the *VD* of *inf*. When *prg1* sends *inf* to *prg2*, the *SL* and *VD* of *inf* are also sent. The parameter receiving *inf* should be associated with the *SL* and *VD* to protect *inf* in *prg2*.

The *SL*s, *VD*s, and join operations of XIFC ensure that sensitive information managed by a program will not be leaked. However, XIFC cannot control the information after it is output. In this case, the operating system should cooperate with XIFC to prevent information leakage. We propose a possible cooperation approach as described below.

For the information output to devices such as screens and printers, the operating system cannot control its access because it cannot control their locations. Accordingly, system managers should control the locations of sensitive output devices. In general, the location placing a sensitive output device should be comparable with the *SLV* of the device. That is, a device with a high *SLV* should be placed in a location in which only high security level persons can be there. Moreover, the location of a sensitive device should better not change during program execution because the *SLV* of an output device is fixed during program execution. When a program is not under execution, device locations can be changed. However, the new location should still be comparable with the *SLV* of the device. If an output device is migrated to a higher (lower) sensitive location, its *SLV* of the device in the program should be increased (decreased) accordingly. This will cause re-compiling of the program.

For the information output to files, the operating system can control its access. To cooperate with XIFC, the operating system should offer a file access interface operated as follows. First, every role appears in the system should be associated with a *SL* similar to that in XIFC. Second, when a role intends to access a file, the operating system compares the *SL*s of the information stored in the file with the role's *SL*. The operating system then retrieves the information accessible by the role from the file. We use the doctor/patient example to explain the necessity of the file access interface. Suppose the case histories of all patients are stored in a file and a patient can access his own case history only. In this case, if a patient intends to access the case history file, the file access interface will retrieve only the patient's case history.

V. Example

We use partial function of a hospital's patient management system as an example to depict the used of XIFC. The function is described below.

In a hospital, the case histories of patients are stored in a file. A doctor can read and change the case history of a patient assigned to him, and can read but not change the case histories of others.

Since a real hospital supports thousands of patients and each patient should be in an independent group, numbers should be used to represent groups. To depict the use of bits, this example assumes only two doctors *dc0* and *dc1* who support six patients *pt0* through *pt5*. We also assume that: (a) *pt0* through *pt2* is assigned to *dc0* and the others to *dc1* and (b) the patients' case histories are stored in the file *CaseHt*. We use an eight bit byte to represent groups. We also use an eight bit byte to represent *SLV*. The following exhibition uses PDL (program design language) to depict the use of XIFC in which a statement started with two asterisks is a XIFC directive. To depict the control of I/O media, we add a file *CastHt_operator*, a keyboard *Kb_dc0* for *dc0*, a screen *Scrn_dc0* for *dc0*, and a screen *Scrn_operator*.

Exhibition 1. A PDL program segment embedding XIFC

*** GROUP bit(8); // Use 8 bits to represent a group number set.*

*** SLV bit(8); // Use 8 bits to represent a security level number.*

*** SL CaseHt (00111111, 00111111, 10000000) // The file CaseHt can be read/write by variables related to patients (i.e., those in groups 0 through 5) and its SLV is 7.*

*** SL CaseHt_operator (10000000, 10000000, 00000100); // The SLV of CaseHt_operator is 2.*

*** SL Kb_dc0 (00000111, , 10000000); // The read group numbers of Kb_dc0 are 0 through 2. Therefore, pt0 through pt2 can read information from Kb_dc0.*

*** SL Scrn_dc0 (, 00111111, 10000000); // The write group numbers of Scrn_dc0 are 0 through 5. Therefore, pt0 through pt5 can write information to Scrn_dc0.*

*** SL Scrn_operator (, 10000000, 00000100); // No patient can write Scrn_operator.*

Variables: newCaseHt_dc0, newCaseHt_dc1, obtainedCaseHt_dc0, obtainedCaseHt_dc1, caseHt_Pt0, caseHt_Pt5, va, vb, vc, vd;

*** SL newCaseHt_dc0 (00000001, 00000001, 10000000); // This variable stores the new case history offered by the doctor dc0 to change the case history of the patient pt0.*

*** SL newCaseHt_dc1 (00100000, 00100000, 10000000); // This variable stores the new case history offered by the doctor dc1 to change the case history of the patient pt5.*

*** SL obtainedCaseHt_dc0 (00111111, ,); // This is a variable for dc0 to read the case histories of patients. The doctor dc0 can read the case histories of every patient.*

*** SL obtainedCaseHt_dc1 (00111111, ,);*

*** SL caseHt_pt0 (00111111, 00000001, 10000000); // This variable stores the case history of pt0. It can be written by variables in group 0 only and can be read by variables in groups 0 through 5.*

*** SL caseHt_pt5 (00111111, 00100000, 10000000); // This variable stores the case history of pt5.*

*** READ; // This directive indicates that the following assignment is a read access.*

obtainedCaseHt_dc0 = caseHt_pt0; // The assignment is allowed according to read group comparison. After the assignment, the SL of obtainedCaseHt_dc0 will be changed to (00111111, 00000001, 10000000) according to the join operation set J1.

*** WRITE; // This directive indicate that the following assignment is a write access.*

caseHt_pt5 = newCaseHt_dc1; // The assignment is allowed according to write group comparison. After the assignment, the SL of caseHt_pt5 will be changed to (00100000, 00100000, 10000000) according to the join operation set J2.

*** XSL caseHt_pt5 (00111111, 00100000, 10000000); // The directive XSL changes SLs.*

readKeyboard(Kb_dc0, caseHt_pt0); // This read is allowed according to group comparison. The SL of caseHt_pt0 will be changed to (00000111, 00000001, 10000000) according to the join operation set J4.

// Before outputting a patient's case history, the SL should be changed for proper protection.

*** XSL caseHt_pt0 (00000001,00000001,10000000);*

writeScreen(Scrn_dc0, caseHt_pt0); // This write operation is allowed according to group and SLV comparisons. The SL of Scrn_dc0 will be unchanged because the SLs of I/O media are fixed.

writeFile(CaseHt, caseHt_pt0); // The statement is allowed.

// Some statements that will be banned are shown below.

obtainedCaseHt_dc0 = caseHt_pt0; // The assignment is allowed. After the assignment, the SL of obtainedCaseHt_dc0 will be changed to (00111111, 00000001, 10000000) according to the join operation set J1.

writeFile(CaseHt_operator, obtainedCaseHt_dc0); // This statement will be banned because the SLV of the file CaseHt_operator (which is 2) is smaller than that of obtainedCaseHt_dc0 (which is 7).

readKeyboard(Kb_dc0, caseHt_pt5); // This statement will be banned because it failed to pass the read group comparison.

writeScreen(Scrn_operator, caseHt_pt0); // This statement will be banned because it failed to pass the write group and SLV comparisons.

// Some assignment statements are shown below.

*** SL va (01000000,01000000,00001000);*

*** SL vb (01000000,01000000,00100000);*

*** SL vc (10000000,10000000,);*

// The variable vd is initially non-sensitive.

vd = va+vb+100; // This statement is allowed because of group comparison. After the assignment, the non-sensitive variable vd becomes sensitive and is associated with the SL (01000000, 01000000, 00100000) according to the join operation set J3.

vd = vc+vd; // This statement will be banned because vc and vd are in different groups.

. . . .

The above exhibition shows the following facts: (a) group comparison dominates the validity of non-output statements (the comparison prevents information corruption) and (b) *SLV* comparison is applied only when information is output (the comparison prevents information leakage).

VI. Proof of Correctness

The primary mission of an information flow control model is preventing information leakage and corruption. To prove information corruption will not occur is difficult because even a data source trusted by a variable may incidentally corrupt the variable. In XIFC, variables in the same write group are considered mutually trusted and write operations can occur among the variables. Although we cannot ensure XIFC prevents every corruption, we can at least say that malicious corruption will not occur. For example, sending a customer's member number to a variable storing his salary will corrupt the variable. XIFC will ban this corruption because the variable storing a customer's member number and that storing his salary are in different write groups. To prove XIFC prevents information leakage, we need the following assumptions:

a. The cooperation among XIFC, the system managers, and the operating system function correctly (see the description near the end of section 4.2). Otherwise, either the system managers or the operating system may leak information.

b. Programmers do not commit errors. Otherwise, unpredictable leakage may occur. For example, misusing the directive *XSL* may cause serious results.

In addition to the assumptions, we also need the following definition.

Definition 4. DEV_{var} = {dev_{var} | dev_{var} is a variable derived from the variable *var*}. For example, the statement "var0 = var + var1;" causes *var0* to become an element of DEV_{var}. A dev_{var} belonging to DEV_{var} is associated with SL_{devvar} and VD_{devvar} in which SL_{devvar} = ($Grw_{devvar}, Gr_{devvar}, Gw_{devvar}, SLV_{devvar}$).

Suppose *var* is a sensitive variable associated with SL_{var} and VD_{var} in which SL_{var} = (Grw_{var}, Gr_{var}, Gw_{var}, SLV_{var}). As we have emphasized, only output operations may cause information leakage. Therefore, the information stored in *var* will not be leaked (i.e., XIFC prevents information leakage) if the following cases are all true.

a. The variable *var* and every dev_{var} in DEV_{var} will not be output to devices or files whose groups are incomparable with those of *var*.

b. The variable *var* and every dev_{var} in DEV_{var} will not be output to devices or files whose *SLV*s are smaller than SLV_{var}.

c. The variable *var* and every dev_{var} in DEV_{var} will not be sent to programs not allowed to receive *var*.

d. None of the above cases will happen to a program reading information from a file output by another one (here we suppose XIFC is embedded in the programs).

Case *d* is necessary because information output to a file may be read and leaked by other programs. In case *a*, *var* will not be output to group incomparable devices or files because group comparison will be performed before the output. As to dev_{var}, the intersections on groups in the join operation sets J1 through J3 ensure that "$Grw_{devvar} \subseteq Grw_{var}$", "$Gr_{devvar} \subseteq Gr_{var}$", and "$Gw_{devvar} \subseteq Gw_{var}$". Therefore, no dev_{var} will be output to devices or files whose groups are incomparable with those of *var*.

In case *b*, *var* will not be output to devices or files with a smaller *SLV* because *SLV* comparison will be performed before the output. As to dev_{var}, the *MAX* function in the join operation sets J1 through J3 ensures that "$SLV_{devvar} \geq SLV_{var}$". Therefore, no dev_{var} will be output to devices or files with *SLV*s smaller than SLV_{var}.

In case *c*, *var* will not be sent to programs not allowed to receive *var* because of *VD* checking before the sending (see item *e* of section 4.2). As to dev_{var}, the intersections on *VD*s in the join operation sets J1 through J3 ensure that "$VD_{devvar} \subseteq VD_{var}$". Therefore, dev_{var} will not be sent to programs not allowed to receive *var*.

In case *d*, the join operation set J5 ensures that the variable *var* reading the information *inf* stored in a file will be associated with the *SL* and *VD* of *inf*. Cases *a* through *c* above ensure that *var* will not leak *inf*. #

VII. Evaluation

We embedded XIFC in C language and implemented a prototype. The prototype is a preprocessor, which changes XIFC directives into C definitions or statements and stores *SL*s and *VD*s. It also adds code to: (a) manage the directives *XSL* and *DECLASSIFY*, (b) check the security of information flows and, (c) do join operations. The runtime overhead of XIFC deviates substantially according to different systems. For example, if a system manages few sensitive variables and bits can represent groups, the runtime overhead should be low because bit operations can be applied. As another example, if a system manages a large number of sensitive variables and numbers should be used to represent groups, the runtime overhead should be high because bit operations cannot apply. Although we still use assembly procedures, the procedures are more time consuming than bit operations. We used the following systems in the experiments to check the runtime overheads of XIFC.

a. An advertising system in which only the real prices are sensitive. In this system, bits can represent groups.

b. A depositing/withdrawing system of a bank. In this system, most information is sensitive. However, since customers cannot access the files storing their information, customer information can be categorized into few groups. Therefore, bits can represent groups.

c. An order management system. In the system, we suppose only the credit card information and the order histories of customers are sensitive. We also suppose that a customer can access his own order history storing in a file. Since the number of customers is generally large and different customers should belong to different groups, numbers should be used to represent groups.

d. A system managing the case histories of patients. In this system, numbers should be used to represent groups. Moreover, most variables are sensitive.

Before evaluating runtime overhead, we first obtained the detection percentage of invalid statements (i.e., statements that may leak information). We required students to inject invalid statements into programs embedding XIFC. The experiments showed that every injected invalid statement was detected. After that, we required students to implement: (a) the about mentioned four systems embedding XIFC and (b) the same systems not embedding XIFC. We then collected the following information: (a) the percentage of sensitive variables during runtime, (b) the runtime of the systems embedding XIFC, and (c) the runtime of the systems not embedding XIFC. Since sensitive variables may become non-sensitive and vice versa according to joins and the *XSL* and *DECLASSIFY* directives, the percentage of sensitive variables is not fixed during runtime. The averaged percentages of sensitive variables for the four systems are respectively between: (a) 5% and 8%, (b) 91% and 95%, (c) 35% and 40%, and (d) 88% and 93%.

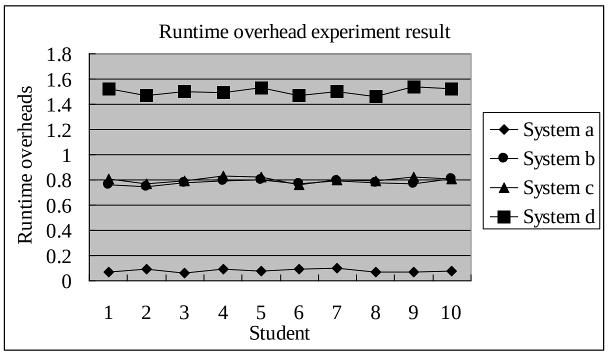

Fig. 1: The experiment data of runtime overhead

To identify the runtime overheads of the systems, the systems embedding and those not embedding XIFC were executed and their runtimes were compared. The experiment result is depicted in Fig. 1. In the figure, a system with a runtime overhead *k* corresponds that the runtime of the system embedding XIFC is *k+1* times the runtime of that without the embedding. Fig. 1 shows that the runtime overhead of system *d* is about 1.5, which is much smaller than those of the models we developed before. The runtime overheads for systems *a*, *b*, and *c* are about 0.08, 0.78, and 0.8 respectively. These overheads are attractive, which implies that the features offered by XIFC are useful. The runtime overheads of the four system deviate substantially because of the factors: (a) the percentage of sensitive variables and (b) the representation of groups (i.e., using bits or numbers).

VIII. Conclusion

Information flow control models can ensure secure database interfaces, ensure secure information flows within an operating system and among distributed operating systems, prevent information leakage during program execution, prevent information leakage in web services, and ensure the security of information flowing forward to and backward from cascading web services. This paper discusses the models that prevent information leakage during program execution and excludes others.

Existing information flow control models offer more or less features. Since only information output may cause leakage, controlling output statements is an important feature. However, our survey reveals that existing models generally ignore this feature. We thus emphasize the importance of the control in this paper. In addition, we also intend to design a precise and low runtime overhead model. To achieve this, we re-examined the features offered by existing models and extracted a set of simple and necessary features to design a new model XIFC (X information flow control). With the assistance of join operations, XIFC only strictly controls output statements and allows most other ones. This reduces runtime overhead. To further reduce the overhead, XIFC only controls information flows involving sensitive variables and/or I/O media. Moreover, XIFC uses bits to represent the security levels of sensitive information and uses assembly procedures to monitor information flows. Our experiments show that XIFC bans every non-secure information flows and the runtime overhead is substantially reduced when comparing with our previous work.

References

[1] Li P. and Zdancewic S. Practical Information-flow Control in Web-based Information Systems. In: 18'th IEEE Computer Security Foundation Workshop, 2005.

[2] Krohn M, Yip A, Brodsky M, Cliffer N, Kaashoek M F, Kohler E, and Morris R. Information Flow Control for Standard OS Abstractions. In: SOSP'07, 2007.

[3] Roy I, Porter D E, Bond M D, McKinley K S, and Witchel E. Laminar: Practical Fine-Grained Decentralized Information Flow Control. In: PLDI'09, 2009.

[4] Zeldovich N, Boyd-Wickizer S, and Mazieres D. Securing Distributed Systems with Information Flow Control. In: 7'th Symposium on Operating System Design and Imoplementation, 2006.

[5] Chou S –C and Huang C –H. An Extended XACML Model to Ensure Secure Information Access for Web Services. Journal of Systems and Software, 2010, 83(1): 77-84.

[6] Chou S –C. Dynamically Preventing Information Leakage for Web Services using Lattice. In: 5'th International Conference on Computer Sciences and Convergence Information Technology (ICCIT), 2010.

[7] She W, Yen I -L, Thuraisingham B, and Bertino E. The SCIFC Model for Information Flow Control in Web Service Composition. In: 2009 IEEE International Conference on Web Services, 2009.

[8] She W, Yen I -L, Thuraisingham B, and Bertino E. Effective and Efficient Implementation of an Information Flow Control Protocol for Service Composition. In: IEEE International Conference on Service-Oriented Computing and Applications, 2009.

[9] Harrison M H, Ruzzo W L, and Ullman J D. Protection in Operating Systems. Communications of the ACM, 1976, 19(8): 461-471.

[10] Olivier M S, van de Riet R P, and Gudes E. Specifying Application-level Security in Workflow Systems. In: 9'th International Workshop on Database and Expert Systems Applications, 1998, 346-351.

[11] Thomas R K and Sandhu R S. Task-Based Authorization Controls (TBAC): A Family of Models for Active and Enterprise-oriented Authorization Management. In: IFIP WG11.3 Workshop on Database Security, 1997.

[12] Myers A and Liskov B. Complete, Safe Information Flow with Decentralized Labels. In: 14'th IEEE Symp. Security and Privacy, 1998, 186-197.

[13] Available on http://en.wikipedia.org/wiki/Discretionary_access_control

[14] Bell D E and LaPadula L J. Secure Computer Systems: Unified Exposition and Multics Interpretation. Available on http://csrc.nist.gov/publications/history/bell76.pdf

[15] Denning D E. A Lattice Model of Secure Information Flow. Comm. ACM, 1976, 19(5): 236-243.

[16] Denning D E and Denning P J. Certification of Program for Secure Information Flow. Comm. ACM, 1977, 20(7): 504-513.

[17] Brewer D F C, and Nash M J. The Chinese Wall Access control policy. In: Proc. 5'th IEEE Symp. Security and Privacy, 1989, 206-214.

[18] Sandhu R S. Lattice-Based Access Control Models. IEEE Computer, 1993, 26(11): 9-19.

[19] Samarati P, Bertino E, Ciampichetti A, and Jajodia S. Information Flow Control in Object-Oriented Systems. IEEE Trans. Knowledge Data Eng, 1997, 9(4): 524-538.

[20] Bertino E, Sabrina de Capitani di Vimercati, Ferrari E, and P. Samarati P. Exception-based Information Flow Control in Object-Oriented Systems. ACM Trans. Information System Security, 1998, 1(1): 26-65.

[21] Ferrari E, Samarati P, Bertino E, and Jajodia S. Providing Flexibility in Information flow control for Object-Oriented Systems. In: 13'th IEEE Symp. Security and Privacy, 1997, 130-140.

[22] Maamir A and Fellah A. Adding Flexibility in Information Flow Control for Object-Oriented Systems Using Versions. International Journal of Software Engineering and Knowledge Engineering, 2003,. 13(3): 313-326.

[23] Yasuda M, Tachikawa T, and Takizawa M. Information Flow in a Purpose-Oriented Access Control Model. In: 1997 International Conf. Parallel and Distributed Systems, 1997. 244-249.

[24] Yasuda M, Tachikawa T, and Takizawa M. A Purpose-Oriented Access Control Model. In: 12'th International Conf. Information Networking, 1998, 168-173.

[25] Tachikawa T, Yasuda M, and Takizawa M. A Purposed-Oriented Access Control Model in Object-Based Systems. Trans. Information Processing Society of Japan, 1997, 38(11): 2362-2369.

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

[27] McIlroy M D and Reeds J A. Multilevel Security in the UNIX Tradition. Software - Practice and Experience, 1992, 22(8): 673-694.

[28] Myers A C and Liskov B. A Decentralized Model for Information Flow Control. In: 17'th ACM Symp. Operating Systems Principles, 1997, 129-142.

[29] Myers A C. JFlow: Practical Mostly-Static Information Flow Control. In: 26'th ACM Symp. Principles of Programming Language, 1999, 228-241.

[30] Myers A and Liskov B. Protecting Privacy using the Decentralized Label Model. ACM Trans. Software Eng. Methodology, 2000, 9(4): 410-442.

[31] Zhang C N and Yang C. An Object-Oriented RBAC model for Distributed System. In: Working IEEE/IFIP Conference on Software Architecture, 2001, 24-32.

[32] Ferraiolo D F, Sandhu S, Gavrila S, Kuhn D R, and Chandramouli R. Proposed NIST Standard for Role-Based Access Control. ACM Trans. Information and System Security. 2001, 4(3): 224-274.

[33] Sandhu R S, Coyne E J, Feinstein H L, and Youman C E. Role-Based Access Control Models. IEEE Computer, 1996, 29(2): 38-47.

[34] Nyanchama M and Osborn S. Modeling Mandatory Access Control in Role-Based Security Systems. Database Security IX: Status and Prospects, 1995, 129-144.

[35] Osborn S. Mandatory Access Control and Role-Based Access Control Revisited. In: Proc. Second ACM Workshop on Role-Based Access Control, 1997, 31-40.

[36] Osborn S, Sandhu R, and Munawer Q. Configuring Role-Based Access Control to Enforce Mandatory and Discretionary Access Control Policies. ACM Trans. Info. Sys. Security, 2000, 3(2): 85-106.

[37] Sandhu R. Role Hierarchies and Constraints for Lattice-Based Access Controls. In: Fourth European Symposium on Research in Computer Security, 1996, 65-79.

[38] Izaki K, Tanaka K, and Takizawa M. Information Flow Control in Role-Based Model for Distributed Objects. In: 8'th International Conf. Parallel and Distributed Systems, 2001, 363-370.

[39] Chou S -C. Embedding Role-Based Access Control Model in Object-Oriented Systems to Protect Privacy. Journal of Systems and Software, 2004, 71(1-2): 143-161.

[40] A. Maamir A, A. Fellah A, and A. Salem A, Controlling Flow in Object-oriented Systems. Journal of Information Assurance and Security, 2008, 2(2): 140-146.

[41] Chou S. –C and Chang C –Y. An Information Flow Control Model for C Applications Based on Access Control Lists. Journal of Systems and Software, 2005, 78(1): 84-100.

[42] Chou S. –C and Chen Y. –C. Managing Role Relationships in an Information Flow Control Model. Journal of Systems and Software, 2006, 79(4): 507-522.

[43] Chou S. –C. Providing Flexible Access Control to an Information Flow Control Model. Journal of Systems and Software, 2004, 73(3): 425-439.

[44] Chou S -C, Liu A -F, and Wu C -J, Preventing Information Leakage within Workflows That Execute among Competing Organizations. Journal of Systems and Software, 2005, 75(1-2): 109-123.

Authors' Profiles

Shih-Chien Chou: Professor in the Department of Computer Science and Information Engineering, National Dong Hwa University, Taiwan. He is major in software engineering, process environment, software reuse, and information flow control.

How to cite this paper: Shih-Chien Chou,"A Partial Ordered Number System for Information Flow Control", International Journal of Information Technology and Computer Science(IJITCS), vol.5, no.4, pp.1-13, 2013.DOI: 10.5815/ijitcs.2013.04.01