I. J. Computer Network and Information Security, 2015, 3, 1-9
Published Online February 2015 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijcnis.2015.03.01

Controlling Information Flows in Net Services with Low Runtime Overhead

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

***Abstract*—**This paper presents the information flow control model NetIFC to prevent information leakage when a net service is being executed. NetIFC offers the following features: (1) it blocks at least statements as possible and (2) it reduces runtime overhead. To achieve the first feature, NetIFC strictly controls output statements because only output may leak information. To achieve the second feature, NetIFC is executed in parallel with a service in different sites to monitor the service. This monitoring style substantially reduce runtime overhead when comparing with embedding a model in a net service.

***Index Terms*—**Information flow, information flow control, information security, information leakage prevention, runtime overhead.

I. Introduction

Current software services are generally executed on the net, such as web services and cloud services. Although the research of cloud computing [1] is really hot, the research of other net applications is still important. Therefore, we collectively call services on the net and those on clouds as net services. Net services offer substantial benefit. However, there are problems to solve. Our research focuses on preventing information leakage during the execution of net services. The prevention can be achieved by *information flow control* [2-15].

According to our survey, information flow control models can be classified into two major categories. The first category uses credentials, attributes, or other mechanism to determine whether a requester can invoke a service. For example, the SCIFC model [12] checks whether a service chain can be successfully invoked through attributes/credentials checking. Models in this category fail to check information flows *within* software. They are considered imprecise because program statements may leak information *during execution*. For example, a malicious account management system may leak user accounts during execution. As to models in the second category, such as the decentralized label model [4-5], they are *embedded* in software to check information flows in the software during execution. Control in this category is precise but induces runtime overhead. For example, using *n* statements to check a statement results in about *n+1* times the normal runtime. In general, models in this category prevent information but fail to discuss runtime overhead.

We developed models in the second category [16]. However, high runtime overhead induced by model embedding bothered us. Thanks to the mass parallelism of net environments, we developed an information flow control model for net services named NetIFC. It is not embedded in services. On the other hand, NetIFC and the net service being monitored are executed in parallel on different sites. According to non-embedding, the runtime overhead is expected to be reduced substantially. NetIFC uses *security level numbers* to prevent information leakage, in which larger security level numbers imply more sensitivity. When designing the model, we identified the following facts.

1. Without considering viruses, worms, and the attack mentioned in [18], only output statements need to be controlled because only output information may be leaked (to persons or other programs). As to other statements, they need not be controlled. Nevertheless, the *join* operation [4] should be used to adjust security level numbers so that leakage can be prevented when variables are output later. Control in this manner simplifies a model. However, existing models are unnecessarily complicated because they generally control every statement.
2. Information exchange is sometimes illegal even with the same data type. For example, salaries with the units of EUR and USD are incomparable. Exchanging incomparable information results in *information corruption*. We use *groups* to prevent the corruption. Information in different groups cannot be exchanged.
3. Embedding a model in a program induces large runtime overhead. We have mentioned this point already.

According to the facts, NetIFC strictly controls output statements and uses join operations to adjust security level numbers for other statements. In addition, NetIFC is *not embedded* in a net service. Instead, it is executed *in parallel* with the service on different sites to monitor the service. According to non-embedding, runtime overhead is expected to be substantially reduced. Note that our research assumes that information transferring on the network is secure and thus skips the problem of cryptography. This paper presents NetIFC.

II. Related Work

Traditional lattice-based model [2-3] is a MAC (mandatory access control). The "can flow" relationship is based on the rules of "no read up" and "no write down" [19], which are generally criticized as too restricted. The decentralized label model [4-5] uses labels to mark the security levels of variables. A label is composed of policies to be simultaneously obeyed. A policy in a label is composed of an owner and zero or more readers that can read the variable. The model in [20] uses ACLs of objects to compute ACLs of executions. A message filter is used to filter out possibly non-secure information flows. Interactions among executions are categorized into five modes, which loosen the restriction of MAC by using different policies for different modes.

Flume [6] 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 transferred among processes as a whole). The secrecy tags prevent information leakage and integrity tags prevent information corruption. Flume also avoids information leaked to untrusted channel (e.g., sockets). The function of Laminar [7] is similar to that of Flume. Nevertheless, the control granularity is detailed to data structures and system resources.

The model Trust-Serv [21] uses state machines to dynamically choose web services at run time. The choosing is a kind of negotiation. It uses trust negotiation [22] to select web services that can be invoked. The kernel component to determine whether a web service can be invoked is *credential*. The model proposed in [23] uses digital credentials for negotiation. It defines strategies for negotiation policies. The model proposed in [24] handles k-leveled invocation of web services. The primary component used in the model is credentials. SCIFC [12] uses various mechanism (such as back-check procedure, carry-along policies, and transformation factors) and algorithms to make sure whether a service chain can be successfully invoked. The objectives of quite a few researches are also about negotiation [25-30]. A negotiation model generally fails to check information flows within software during execution. The models are considered imprecise as mentioned in section 1.

The model proposed in [8] uses X-GTRBAC [31] to control the access of web services. X-GTRBAC can be used in heterogeneous and distributed sites. Moreover, it applies TRBAC [32] to control the factor related to time. The model in [9] determines whether a composition of web services can be invoked. It offers a language to describe policies, which check whether a composition of web services can be invoked. The model in [10] uses RBAC (role-based access control) concept [33-34] to define policies of accessing a web service. It is a two-leveled mechanism. The first level checks the roles assigned to requesters and web services. An authorized requester is a candidate to invoke a web service. The second level uses parameters as attributes and assigns permissions to the attributes. An authorized requester can invoke a web service only when it possesses the permissions to access the attributes. The model proposed in [11] allows requesters to use PMI (privilege management infrastructure) for managing the checking, retrieval, and revocation of authentication. The model also uses an RBAC-based web service access control policy to determine whether a requester can invoke a web service.

The model in [13] identifies dependencies among I/O parameters of services to decide whether service invocations will leak information. The model in [14] offers functions to check intra-component information flows through the JIF language [35]. It also defines protocols to check inter-component information flows. Although both intra- and inter-components information flows are checked, JIF embeds decentralized labels in a program. The embedding will induce runtime overhead. The model in [15] applied Chinese Wall [36] to control information flows in the IaaS level. As to services in the SaaS level, the model cannot control their information flows. In the recent survey of [37], the authors especially emphasize the importance of data privacy in a cloud computing environment. They believed that information flow control is a good solution for the problem. However, the article did not propose a concrete information flow control model for cloud environments.

As stated in section 1, existing models can be classified into two categories. Models such as [12] belong to the first one, which fail to check information flows within a service and results in imprecise control. Models such as [14] belong to the second one. Although they are precise, they may induce large runtime overhead.

III. NetIFC

Before defining NetIFC, we describe information leakage. *Information leakage occurs when persons or other software illegally obtain sensitive information from a software system*. In general, persons can obtain information from files or output devices and software can obtain information from files. Information leakage may thus happen when sensitive information is output.

When designing NetIFC, we let the control granularity be detailed to variables. That is, NetIFC controls information flows among variables. We need this granularity because different variables carry information with different sensitivities. According to the description in section 1, a *group* and a *security level number* facilitate preventing information corruption and leakage, respectively. We thus combine them into a *security level*. That is, a group *Gp* and a security level number *Slv* is combined into a *security level*: "(*Gp, Slv*)". Since the control granularity is detailed to variables, *every variable is associated with a security level*. To facilitate normal execution of a service, the security levels of non-sensitive variables are set "(Global, -1)". The setting allows non-sensitive variables to flow anywhere because *intersecting a set with "Global" obtains the set itself.* A non-sensitive variable may become sensitive and vice versa during runtime. For example, suppose: (a) the initial security

level of the variable *b* is (*Gp1*, *Slv1*) and (b) the variable *a* is initially non-sensitive. Then, after the statement "a=b;" is executed, the security level of the variable *a* is adjusted to be the same as that of *b*. This causes *a* to become sensitive.

Since sensitive information operated by software may be leaked only when it is output, every file and output device should also be associated with a security level. In general, the security level of a device is decided by its location. For example, if an output device is placed in the office of a manager, its security level is set according to the importance of the manager. If the location of a device is changed, its security level may be changed. NetIFC cannot handle the change because only persons know the change. As to files, their security levels also cannot be adjusted because the adjustment will affect the access of the files. For example, if the security level number of a file is 3, it can be accessed by a user possessing a privilege to access files whose security level number is 3 or lower. If the security level number of a file is adjusted to be 4, the user can no longer access the file.

The kernel concepts of NetIFC have been described above. Below we give a definition to NetIFC.

Definition 1. *NetIFC* = (*SL*, *VAR*, *ID*, *OD*, *FI*, *Rules*), in which

1. *SL* is the set of security levels. Every component in the set is composed of a group and a security level number. That is, *SL* = {(*Gp*, *Slv*)$_i$ | *Gp* is a group, *Slv* is a security level number, and (*Gp*, *Slv*)$_i$ is a security level}.
2. *VAR*, *ID*, *OD*, *FI* are the sets of variables, input devices, output devices, and files used by a service, respectively. Every component in the sets is associated with a security level.
3. *Rules* are NetIFC's information control rules. To facilitate describing the rules, we need the following definition.

Definition 2. *IFS* is "a statement that will result in information flow", including the statements of (a) assignment, (b) input from devices or files, (c) output to devices or files, (d) invoking subroutines or methods, and (e) invoking other services. Invoking other services may result in chains mentioned in SCIFC [12]. Since only security levels are needed to control information flows, operators in a statement such as "+" and "-" can be ignored here. Therefore, the following definition of *IFS* does not contain operators.

IFS = {(*Des*, Gp_{Des}, Slv_{Des}), {(*Source*, Gp_{Source}, Slv_{Source})$_i$} | (*Des*, Gp_{Des}, Slv_{Des}) is the destination of the information flow. (*Source*, Gp_{Source}, Slv_{Source})$_i$ is a source for the flow. The destination and every source are associated with a security level "(*Gp*, *Slv*)".}

Since NetIFC controls only output statements, we categorize statements into output statements (*OPSs*) and non-output statements (*NOPSs*). NOPSs that may induce information flows include *assignments*, *input statements*, *invocation statements* (invoking subroutines or object methods), and *statements that invoke other services*. OPSs include outputting to devices or files. As mentioned above, NetIFC strictly controls OPSs. Non-secure OPSs will be blocked. As to NOPSs, they can always be executed. However, join operations will be executed to adjust security levels. The adjustment prevents information leakage when output occurs later. Suppose *a* is derived from the set of variables {x_i | x_i is a variable}. Also suppose that the group of *a* and every x_i are the same. Then, the join operation adjusts the *Slv* of *a* to be MAX(Slv_{x_i}), in which the function MAX extracts the maximum value.

NOPSs other than assignments can be considered variations of assignments. For example, an input statement retrieves a value from a device/file and *assigns* the value to the variable. Other NOPs can be considered similarly. According to the description above, NetIFC offers a rule to control OPSs and a rule to do join for NOPSs. The rules are listed below.

Rule 1. For an OPS "((*Odes*, Gp_{Odes}, Slv_{Odes}), {(*Source*, Gp_{Source}, Slv_{Source})$_i$ })", in which $Odes \in (OD \cup FI)$ (check Definition 1 for the definition of *OD* and *FI*), both the condition "$(\cap_i Gp_{Source_i} \cap Gp_{Odes} \neq \phi)$" and "$MAX(Slv_{Source_i}) \leq Slv_{Odes}$" should be true. The first condition requires that the destination and the sources should be in the same group. It prevents exchanging incomparable information. The second condition prevents information leakage.

Rule 2. For an assignment "(*Des*, Gp_{Des}, Slv_{Des}), {(*Source*, Gp_{Source}, Slv_{Source})$_i$}", it can be executed if the condition "$(\cap_i Gp_{Source_i} \cap Gp_{Des} \neq \phi)$" is true. The condition prevents exchanging incomparable information as mentioned above. After statement execution, the following *join* operation adjusts the security level of the destination.

$Slv_{Des} = MAX(Slv_{Source_i})$

IV. Proof of Correctness

Information leakage may occur during output. The output statements are listed below.

1. ((*Od*, Gp_{Od}, Slv_{Od}), {(*Source*, Gp_{Source}, Slv_{Source})$_i$ }): This type of statement outputs the information derived from a set of variables to an output device *Od*.
2. ((*Fi*, Gp_{Fi}, Slv_{Fi}), {(*Source*, Gp_{Source}, Slv_{Source})$_i$ }): This type of statement outputs the information derived from a set of variables to a file *Fi*.

Case 1. The output to device statement "((*Od*, Gp_{Od}, Slv_{Od}), {(*Source*, Gp_{Source}, Slv_{Source})$_i$ })" will not leak information.

Proof. Information output to a device may only leak to

persons. Suppose person *P* possesses a security level number SLV_p and $SLV_p < Slv_{Od}$. Rule 1 allows the output only when $MAX(Slv_{Source_i}) \le Slv_{Od}$ (suppose *Od* and all sources are in the same group). If the output is allowed, *P* cannot access information output to *Od* because $SLV_p < Slv_{Od}$. Since $Source_i$ derives the information output to *Od*, no $Source_i$ will be leaked to *P*.

Case 2. The output to file statement ((*Fi*, Gp_{Fi}, Slv_{Fi}), {(*Source*, Gp_{Source}, Slv_{Source})$_i$ }) will not leak information.

Proof. Information output to a file may leak to persons or software. Rule 1 allows the output only when $MAX(Slv_{Source_i}) \le Slv_{Fi}$ (suppose *Fi* and all sources are in the same group). Case 1 states that no $Source_i$ will be leaked to a person *P* with a security level number SLV_p and $SLV_p < Slv_{Fi}$. On the other hand, if $Software_i$ intends to read the information of *VAR* from *Fi* to the variable VAR_1, Rule 2 causes $Slv_{var} = Slv_{var_1}$. If $Software_i$ outputs VAR_1 to a device, Case 1 states that VAR_1 will not be leaked to a person. If VAR_1 is output to another file, the proof goes back to the beginning of this proof. That is, the proof is endless. However, we know that the information of a sensitive variable may be: (a) operating by a net service, (b) output to a device, or (c) output to a file. Information operating by a net service will not be leaked without output. Information output to a device will not be leaked (see the proof of Case 1). As to the information output to a file, it will not be leaked to persons. Since NetIFC handles information flows within a limited area of the net, no information leaking to persons corresponds to no information leakage. Therefore, Case 2 is true.

V. Runtime Overhead Reducing

Existing information flow control models are generally embedded in a software system. This induces large runtime overhead. To reduce the overhead, NetIFC is not embedded in a net service. Instead, it is executed in parallel with the service on different sites. Fig. 1 shows the concept. The checking mechanisms of NetIFC use Rules 1 and 2 in section 3 to check information flows.

Since NetIFC checks the information flows of a service, it is impossible for the model and the service to be independent. They might need to synchronize under certain circumstances (this results in the "synchronization data structure" in Fig. 1). We use the following C program segment to explain this.

```
. . .
a=b+c+e;
d=e+f;
g=h+i;
j=a+d;
k=g+j;
l=m+n;
printf("%d",i+g);
. . .
```

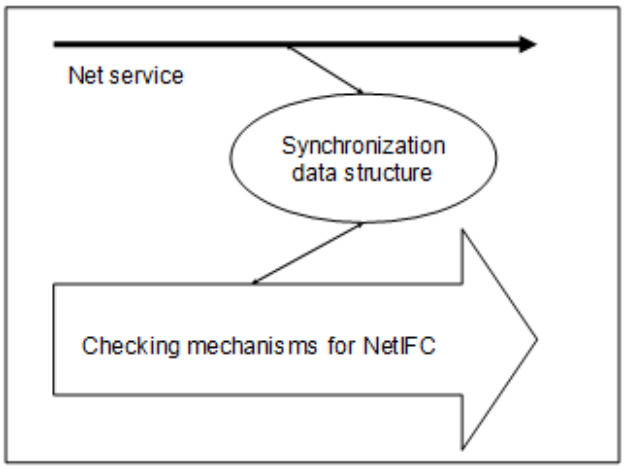

Fig 1. Net service and NetIFC are executed in parallel

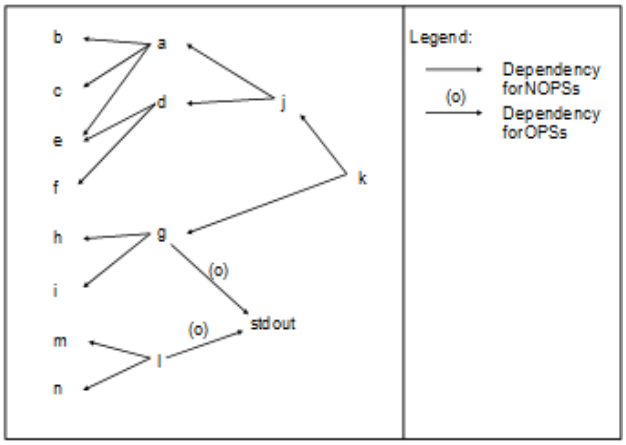

Fig 2. Variable dependent relationships

In the program segment, the security level of variable *a* can be determined only when the security levels of variables *b*, *c*, and *e* are available (enabled). We say that variable *a* depends on variables *b*, *c*, and *e*. According to the dependency, a variable dependency structure can be established. Fig. 2 show the structure for the program segment above. In the figure, there are dependencies for OPSs and those for NOPSs. Rule 1 in section 3 will be applied to check the security of information flows for OPSs while Rule 2 for NOPSs. With the assistance of the structure, a net service and NetIFC can be executed in parallel with necessary synchorization.

When checking a program deeper, Fig. 2 cannot solve everything. For example, a variable can be used more than once. As another example, how can the constructs of selection (if statement) and repetition (loop) be handled? Consider the following program segment:

```
...
a=a+b;
b=b+c;
a=a+b;
printf("%d",a);
…
```

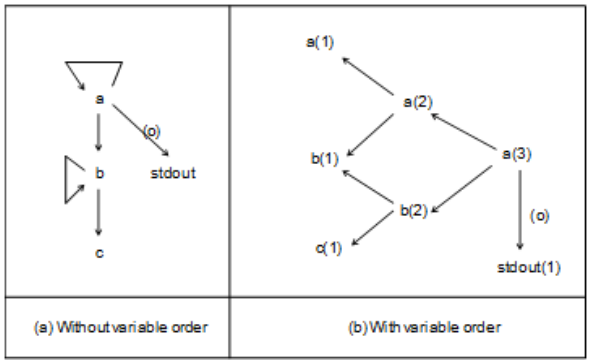

Fig 3. The original dependency structure and the reformed one

If Fig. 2 is used in the program segment, Fig. 3(a) will be obtained. In this case, it is difficult to decide when should variables *a* and *b* be enabled and which statement is being executed. For example, in the program segment above, variables *a* and *b* should be enabled for statements 1 and 3, but it is difficult to distinguish whether statement 1 or statement 3 is being executed. We thus reform Fig. 3(a) into 3(b), in which the appearing order of variables is explicitly shown using sequence numbers within parentheses. With the number, the variable enabling and disabling procedure can be operated normally. For example, when *a(1)* and *b(1)* are enabled, the first "a=b+c;" can be executed. On the other hand, when *a(2)* and *b(2)* are enabled, the second "a=b+c;" can be executed. Fig. 3(b) thus remedies the shortcoming of Fig. 2.

To handle the selection and repetition constructs, variable values should be known. We use the following program segment for the explanation.

```
. . .
a=b+c+e;
d=e+f;
if((a+b)>0) g=h+i;
else g=j+k;
while((d+j)>0){
   b=d-k;
   a=j+k;
   d--;
}
c=a+b;
l=m+n;
o=p+q;
r=l+o;
. . .
```

The variable dependency relationships of the program segment are shown in Fig. 4. In the figure, the selection construct causes variable *g* to appear in two statements that depend on different conditions. When a condition is true, the join operations of the variables depending on the condition will be enabled. That is, the statements following the "if" keyword will be executed and those following the "else" keyword will be skipped. The figure uses different type of arrows to show different types of variable dependencies. Solid arrows are used to perform the join operation for a variable while dash ones are for conditions.

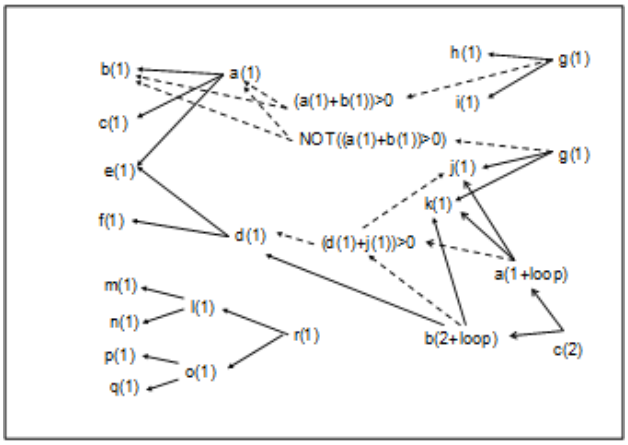

Fig 4. Variable dependencies

Fig. 4 also shows the variable dependencies for the repetition construct. In the dependencies, the sequence numbers of the statements within the loop is shown as "n+loop" because we do not know when the loop will end. During information flow checking, if NetIFC identifies the security levels of variables within a loop will not change even when the loop is not finished, it will skip other checking and proceeds to the next statement (e.g., the "c=a+b" statement after the loop). Note that the security checking process of the assignment statements for the *l*, *o*, and *r* are not affected by the branch and repetition constructs because no variable dependent relationship exists. Therefore, selections and repetitions may or may not affect information flow checking of statements following them.

To show the rules of establishing variable dependencies, we need the following symbols: (1) $\mathcal{D}(i, j)$ depicts that variable *i derives* variable *j*, (2) $\mathcal{O}(i, j)$ depicts that variable *i* outputs to device/file *j*, (3) $\mathcal{D}_c(i, j)$ and $\mathcal{D}_{nc}(i, j)$ depicts that variable *i depends on* variable *j* for *OPSs* and *NOPSs*, respectively, (4) $\mathcal{E}(C, A)$ depicts that if condition *C* is true, block *A* is executed, (5) $\mathcal{R}(C, A)$ depicts that if condition *C* is true, block *A* is repeated, (6) $\mathcal{V}(C)$ is the set of variables within condition *C*, (7) $\mathcal{D}_{nc}(i, C)$ depicts that variable *i* depends on condition *C*, and (8) $\mathcal{D}_{nc}(C, i)$ depicts that condition *C* depends on variable *i*. The dependency establishment rules are shown below:

1. $\mathcal{D}(a(j), b(i)) \Rightarrow \mathcal{D}_{nc}(b(i+1), a(j))$
2. $\mathcal{O}(a(j), Om(i)) \Rightarrow \mathcal{D}_c(Om(i+1), a(j))$
3. In a selection, either $\mathcal{E}(C, A)$ or $\mathcal{E}(NOT(C), B)$. In either case, rules 1 through 4 listed here establishes variable dependencies in blocks *A* and *B*. Moreover, the following sub-rules are used to establish dependencies for condition *C*.

   3.1. $\forall a \in V(C) \wedge D(a(j), b(i))$: $D_{no}(C, b(i))$
   3.2. $\forall a \in V(C) \wedge D(b(i), a(j))$: $D_{no}(b(i), C)$

4. In a repetition construct, $\mathcal{K}(C, A)$. Rules 3.1 and 3.2 are applied to condition C, and the following sub-rules are applied for block A.

   4.1. $\mathcal{D}(a(j), b(i)) \Rightarrow \mathcal{D}_{nc}(b(i+loop), a(j))$
   4.2. $\mathcal{O}(a(j), \mathrm{Om}(i)) \Rightarrow \mathcal{D}_{c}(Om(i+loop), a(j))$

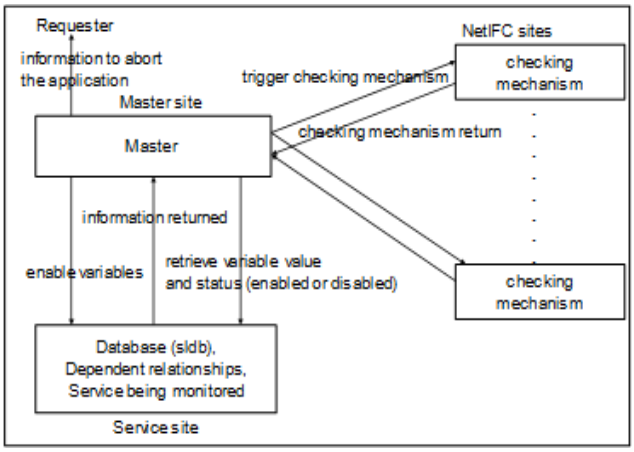

Fig 5. Implementation architecture NetIFC

The implementation architecture of NetIFC is shown in Fig. 5, which is similar to MapReduce [19]. The net service being monitored is executed in the *service site* and NetIFC is executed in others. The data used by the service and the variable dependent relationships are stored in the database "sldb". When a service is being executed, if an output occurs, the variable to be output must be ready (i.e., the security of the output statement must be ensured). With this, a ready flag for the variable should be checked before output. NOPSs are executed without checking readiness. However, if the checking of "within the same group" fails (see Rule 2 in section 3), the service will be aborted.

Fig. 5 shows that the *master site* polls the status (enabled or disabled) of the variables needed by the checking mechanisms. As long as the status of the variables needed by a checking mechanism is enabled, the master requires a site from the net environment to execute a copy of the checking mechanism. After the checking, the site is released. With this design, the master can require multiple sites to execute checking mechanisms of NetIFC in parallel with the service (the larger the net area, the more available sites for the checking mechanisms).

To describe the operations of NetIFC, we need the following symbols: (1) $\mathcal{V}(service)$ is the set of variables appear in a service, (2) $\mathcal{D}_{ep}\mathcal{S}(v)$ and $\mathcal{D}_{ep}\mathcal{V}(v)$ are the sets of variables that the variable v depends on its security levels (solid arrows in Fig. 2) and its values (dash arrows in Fig. 2), respectively, (3) $_{set}\mathcal{D}_{is}(v)$, $_{set}\mathcal{E}_{nb}(v)$, and $_{set}\mathcal{R}_{dy}(v)$ respectively disable the variable v, enables the variable v, and sets the variable v ready, (4) $\mathcal{E}_{nb}(v)$ and $\mathcal{R}_{dy}(v)$ show that the variable v is enable and ready, respectively, (5) $\mathcal{R}_1(v)$ and $\mathcal{R}_2(v)$ mean that the variable v passes the requirements of Rule 1 and Rule 2 in section 3, respectively, (6) $\mathcal{A}_{bort}(service)$ aborts the executing service. The execution logic of NetIFC is shown below:

1. $\forall v \in \mathcal{V}(service)$: $_{set}\mathcal{D}_{is}(v)$
2. $\forall v \in \mathcal{V}(service) \wedge \mathcal{D}_{ep}\mathcal{S}(v) = \phi \wedge \mathcal{D}_{ep}\mathcal{V}(v) = \phi$: $_{set}\mathcal{E}_{nb}(v)$, $_{set}\mathcal{R}_{dy}(v)$
3. $\exists v \in \mathcal{V}(service), \forall v_i \in \mathcal{D}_{ep}\mathcal{S}(v) \wedge \mathcal{E}_{nb}(v_i)$: if $\mathcal{R}_2(v)$ then $\mathcal{E}_{nb}(v)$ else $\mathcal{A}_{bort}(service)$
4. $\exists v \in \mathcal{V}(service), \forall v_i \in \mathcal{D}_{ep}\mathcal{V}(v) \wedge \mathcal{R}_{dy}(v_i)$: if $\mathcal{R}_1(v)$ then $_{set}\mathcal{E}_{nb}(v)$, $_{set}\mathcal{R}_{dy}(v)$ else $\mathcal{A}_{bort}(service)$

VI. Evaluation

To evaluate the performance of NetIFC, we used 10 PCs to simulate the architecture in Fig. 5, in which one of them simulates the master site, another simulates the service site, and the others simulate the NetIFC checking mechanisms.

Although a service and the checking mechanisms for NetIFC can be executed in parallel, the service may delay when intending to output unchecked (unready) variables. The case occurs when the checking mechanisms fail to finish checking the variables before the output statement. To check in details the runtime overhead of NetIFC, we write programs by controlling the length of variable dependency path (e.g., "k -> j -> a -> b" in Fig. 2 is a variable dependency path) and controlling the runtime of the programs. When we need a large runtime, we write programs with loops that execute as many times as needed. As shown in Fig. 5, the runtime overhead of NetIFC is caused by the time consumed by the master, that by the network transfer, and that by the checking mechanisms. For each checking operation, the execution time includes: (1) the master execution time, (2) the checking mechanism execution time, and (3) five network overheads. Fig. 5 shows that the network overheads include: (1) that for master to retrieve variable value and status, (2) that for master to receive variable value and status, (3) that for master to trigger checking mechanism, (4) that for master to receive checking mechanism return, and (5) that for master to enable variables. When evaluating network overhead, we record the time *t1* of a site when sending a message to another site and record the time *t2* when the sender receives a return message. The receiver returning a message to the sender should tells the sender the time *t3* consumed by the processes in the receiver. The sender then calculates network overhead using the formula "*t2 - t1 - t3*".

In the experiments, we want to know the relationships between runtime overhead and the following factors:

1. The network overhead. This factor is closely related to the runtime of a program. If the runtime of a program is too short, network overhead may dominate the runtime overhead.
2. The length of variable dependency path. If the length is too large, there may be few checking

mechanisms can execute in parallel. This may cause large runtime overhead.

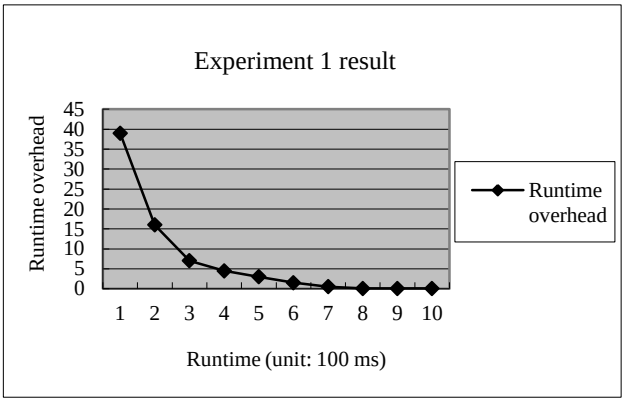

Fig. 6. Experimrnt 1 result

We design experiments to check those we intend to know. The experiment results are obtained from the average of multiple programs' execution results. The first experiment checks the relationships between program runtime and NetIFC runtime overhead. The result is shown in Fig. 6. The value in the X dimension of the figure should be multiplied by 100 ms (i.e., the unit of the X dimension is 100 ms). The figure shows that the network overhead dominates the runtime overhead when program execution time is smaller than about 700 ms.

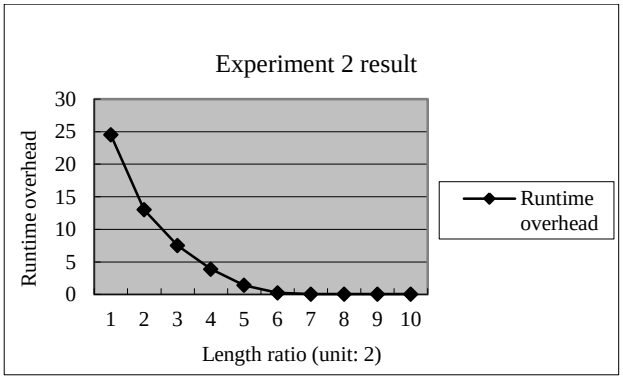

Fig. 7. Experimrnt 2 result

The second experiment checks the relationship between the length of variable dependency path and NetIFC runtime overhead. To eliminate the effect of network overhead, the runtime of programs in this and other experiments are all longer than 1 minute. In this experiment, the ratio between the longest execution thread of a program and the maximum length of the variable dependency path are expected to dominate the runtime overhead. That is, if the longest execution thread of a program is *m* and the maximum length of the variable dependency path is *n*, the ratio *m/n* should be the dominating factor in this experiment. Fig. 7 shows the result of the second experiment. The value in the X dimension of the figure is the ratio mentioned above and it should be multiplied by 2 (i.e., the unit of the X dimension is 2). The figure shows that the runtime overhead can be reduced to *almost zero* if the longest dependency path is smaller than about one twelfth of the longest execution thread of a program. Here we use "almost zero" because every output statement still has to check whether the variables to output are enabled. We will further discuss this in the following experiment.

The third experiment checks the runtime overhead of programs controlled by NetIFC. To eliminate the effect of variable dependency, every variable dependency path is smaller than 4. Fig. 8 shows the result of the third experiment. The value in the X dimension of the figure is the ratio of output statements in a program and it should be multiplied by 0.1% (i.e., the unit of the X dimension is 0.1%). Ideally, the runtime overhead should be zero. However, since variable flags should be checked for the output statement, runtime overhead will happen according to multiple memory access. Nevertheless, the overhead is small, as sown in Fig. 8.

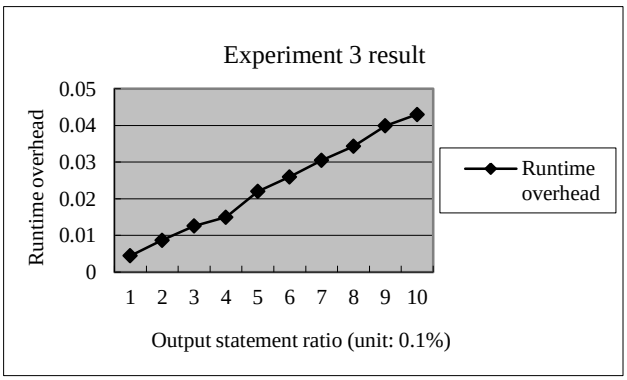

Fig. 8. Experimrnt 3 result

Since NetIFC controls information flows for net services, dishonest service providers may steal output information from physical hardware. We are currently finding solution(s) for the problem.

VII. Conclusion

Information security is a crucial problem for net services. The security covers many research areas. Our research focuses on preventing information leakage during service execution. The prevention can be achieved by information flow control models. We developed NetIFC to control information flows for net services. NetIFC uses groups to prevent exchanging incomparable information and security levels to prevent information leakage. Since our research excludes virus and worms, only output information may be leaked. NetIFC thus strictly controls output statements but allows others. However, NetIFC uses the join operation to adjust the security levels of variables to prevent information leakage when output occurred later.

If NetIFC should be embedded in software services, it may induce large runtime overhead. We thus applied the mass parallelism of the net environment to implement NetIFC. In the implementation, a net service can be executed in parallel with NetIFC. However, the service

and NetIFC should synchronize using variable dependency relationships.

We simulated NetIFC to check its performance. We identified that network overhead and variable dependencies may induce runtime overhead. Our experiments showed that NetIFC does prevent information leakage and found that: (1) if the program runtime is shorter than about 700 ms, the dominated factor of runtime overhead is the network overhead, and (2) if the program runtime is long enough, the dominated factor of NetIFC runtime overhead is the longest variable dependency path. If the longest path is smaller than one twelfth of the longest execution thread of a service, runtime overhead can be reduced to almost zero.

REFERENCES

[1] L. M., Vaquero, L. Rodero-Merino, J., Caceres, M., Lindner, "A Break in the Clouds: Towards a Cloud Definition", *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, pp. 50-55, 2009.

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

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

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

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

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

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

[8] Bhatti, R., Bertino, E., Ghafoor, A., 2004. A Trust-based Context-aware Access Control Model for Web Services. *IEEE ICW'04*, 184 – 191.

[9] Seamons, K. E., Winslett, M., Yu, T., 2001. Limiting the Disclosure Access Control Policies during Automated Trust Negotiation. *Network and Distributed System Security Symposium.*

[10] Wonohoesodo R., Tari, Z., 2004. A Role Based Access Control for Web Services. *Proceedings of the 2004 IEEE International Conference on Service Computing*, 49-56.

[11] Shen, H. -B., Hong, F., 2006. An Attribute-Based Access Control Model for Web Services. *IEEE International Conference on Parallel and Distributed Computing Applications and Technologies (PDCAT'06)*, 74-79.

[12] W. She, I. -L. Yen, B. ThuraiSingham, E. Bertino, "The SCIFC Model for Information Flow Control in Web Service Composition", *2009 IEEE International Conferences on Web Services*, pp. 1-8, 2009.

[13] W. She, I. -L. Yen, B. ThuraiSingham, and S. –Y. Huang, "Rule-Based Run-Time Information Flow Control in Service Cloud", *2011 IEEE International Conferences on Web Services*, pp. 524-531, 2011.

[14] L. Sfaxi, T. Abdellatif, R. Robbana, and Y. Lakhnech, "Information Flow Control of Component-Based Distributed Systems", *Concurrency and Computation: Practice and Experience*, available on http://www.bbhedia.org/robbana/Wiley.pdf.

[15] R. Wu, G. –J., Ahn, H. Hu, and M. Singhal, "Information Flow Control in Cloud Computing", *Proceedings of the 6th International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 2010.

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

[17] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", *OSDI 2004*, available on http://static.usenix.org/event/osdi04/tech/full_papers/dean/dean.pdf.

[18] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds", *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pp. 199-212, 2009.

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

[20] Samarati, P., Bertino, E., Ciampichetti, A., and Jajodia, S., 1997. Information Flow Control in Object-Oriented Systems. *IEEE Trans. Knowledge Data Eng.*, vol. 9, no. 4, 524-538.

[21] Samarati, P., Bertino, E., Ciampichetti, A., and Jajodia, S., 1997. Information Flow Control in Object-Oriented Systems. *IEEE Trans. Knowledge Data Eng.*, vol. 9, no. 4, 524-538.

[22] Yu, T., Winslett, M., Seamons, K., 2003. Supporting Structured Credentials and Sensitive Policies through Interoperable Strategies for Automated Trust Negotiation. *ACM Transactions on Information and System Security*, vol. 6, no. 1, 1-42.

[23] Koshutanski, H., Massacci, F., 2005. Interactive Credential Negotiation for Stateful Business Processes, *Lecture notes in computer science*, 256-272.

[24] Mecella, M., Ouzzani, M., Paci, F., Bertino, E., 2006. An Access Control Enforcement for Conversation-based Web Services. *International World Wide Web Conference*, 257-266.

[25] Bertino, E., Squicciarini, A. C., Martino, L., Pacim, F., 2006. An Adaptive Access Control Model for Web Service. *International Journal of Web Service Research*, vol. 3, no. 3, 27-60.

[26] Paurobally, S., Jennings, N. R., 2005. Protocol Engineering for Web Service Conversations. *Engineering Applications of Artificial Intelligence*, vol. 18, no. 2, 237-254.

[27] Srivatsa, M., Iyengar, A., Mikalsen, T., Rouvellou, I., Yin, J., 2007. An Access Control System for Web Service Compositions. *2007 IEEE International Conference on Web Services*, 1-8.

[28] Ardagna, C. A., Damiani, E., 2006. A Web Service Architecture for Enforcing Access control Policies. *Electronic Notes in Theoretical Computer Science*, vol. 142, 47-62.

[29] Koshutanski H., Massacci, F., 2003. An Access Control Framework for Business Processes for Web Service. *ACM Workshop on XML Security*, 15-24.

[30] Sirer E. G., Wang, K., 2002. An Access Control Language for Web Services. *Proceedings of the seventh ACM symposium on Access control models and technologies* (*SACMAT'02*), 23-30.

[31] Bhatti, R., Ghafoor, A., Bertino, E., Joshi, J. B. D., 2005. X-GTRBAC: An XML-Based Policy Specification Framework and Architecture for Enterprise-Wide Access

Control. *ACM Transactions on Information and System Security (TISSEC)*, Vol. 8, No. 2, 187 – 227.
[32] Bertino, E., Bonatti P. A., Ferrari, E., 2001. TRBAC: A Temporal Role-Based Access Control Model. *ACM Transactions on Information and System Security*, Vol. 4, No. 3, 191 – 233.
[33] Sandhu, R. S., Coyne, E. J., Feinstein, H. L., Youman, C. E., 1996. Role-Based Access Control Models. *IEEE Computer*, vol. 29, no. 2, 38-47.
[34] R. Sandhu, D. F., Ferraiolo, and D. R., Kuhn, D. "The NIST Model for Role Based Access Control: Toward a Unified Standard", *5th ACM Workshop Role-Based Access Control*. pp. 47–63, 2000
[35] JIF website, "Jif: Java + information flow", available on http://www.cs.cornell.edu/jif/
[36] Brewer, D.F.C., Nash, M.J., 1989. The Chinese Wall Security Policy. In: Proceedings of the 5'th IEEE Symposium on Security and Privacy, 206-214.
[37] J. Bacon, D. Eyers, T. F. J. –M. Pasquier, J. Singh, and P. Piezuch, "Information Flow Control for Secure Cloud Computing", *IEEE Trans. Network and Service Management*, 11(1), pp. 76-89, 2014.

Authors' Profiles

Shih-Chien Chou is a 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,"Controlling Information Flows in Net Services with Low Runtime Overhead", IJCNIS, vol.7, no.3, pp.1-9, 2015.DOI: 10.5815/ijcnis.2015.03.01