Dynamic Information Flow Control Architecture for Web Applications

Sachiko Yoshihama[1], Takeo Yoshizawa[1], Yuji Watanabe[1], Michiharu Kudoh[1], and Kazuko Oyanagi[2]

[1] IBM Tokyo Research Laboratory, Yamato, Kanagawa, Japan
{sachikoy,ytakeo,muew,kudo}@jp.ibm.com
[2] Institute of Information Security, Yokohama, Kanagawa, Japan
oyanagi@iisec.ac.jp

Abstract. In typical Web applications, the access control at the database management system is not effective due to the dependency on application behavior. That is, once the information is retrieved, a careless application can easily leak the information to undesirable parties. In addition, database accounts are often shared for multiple Web users in order to allow connection pooling. We propose DIFCA-J (Dynamic Information Flow Control Architecture for Java), to keep track of and control fine-grained information propagation through execution of the program. DIFCA-J allows controlling the information flow at run-time, without needing to modify the source code of the target application or the Java VMs.

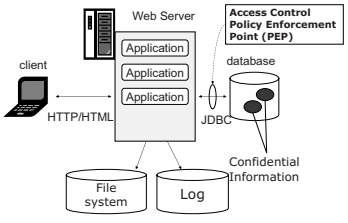
1 Introduction

In a typical three-tier Web application server (Figure 1), sensitive information, such as user's personal information or credit card numbers, is stored in a database. Access to the database is controlled at the database management system, and limited to only authorized users. However, such control is not sufficient to protect sensitive information.

First, after the application retrieves sensitive information from the database, it is up to the application code how to handle such information. An erroneous application may carelessly release such information to an undesirable destination. Furthermore, a single database table may contain data that belong to different classification levels. Some database systems support fine granular access control, but again, once the data is retrieved from the database, protection of the data depends on the application behavior.

Second, many of today's Web applications use a single database account for processing requests from multiple Web users, in order to effectively reuse database connections through connection pooling to optimize performance. The access control at the database is not effective when the same database account is used for all users. For example, a credit card number for a user A may be presented to user B due to a bug in the application, because the single database account cannot distinguish between users.

J. Biskup and J. Lopez (Eds.): ESORICS 2007, LNCS 4734, pp. 267–282, 2007.




```
protected void doGet(HttpServletRequest req,
  HttpServletResponse res) throws ... {
String user = req.getParameter("user");
String item = req.getParameter("item");
pw = new PrintWriter(res.getOutputStream());
...
String credit = getCreditCardInfoFromDB(user);
boolean b = processPurchase(user, item, credit);
if(b){ // succeeds
    pw.println("Purchase Succeeded: <br/>");
    pw.println("Name: " + user + "<br/>");
    pw.println("Item: " + item + "<br/>");
    pw.println("Credit Card: " + credit );
}else{ // failed
    printlog("Invalid credit card: " + credit);
} ... }
```

Fig. 1. Three-tier Web App. Example **Fig. 2.** Problematic Servlet Code Example

Figure 2 shows a sample application in which a program bug can cause undesirable information flows. This is a simple example of an on-line shop servlet, which receives a user name and the item name from an HTTP request, and processes purchase request using the user's credit card number stored in the database. The `processPurchase` method checks the validity of the credit card number, and stops process by returning false when any problem is detected (e.g., the credit card is expired). We assume that the information received from the user via an HTTP request is not confidential, while credit card numbers in the database are confidential. (We assume that a proper user authentication takes place in advance, and the communication channel is protected by SSL or TLS, and thus the information in the HTTP request can be trusted.)

When looking at this program from the aspect of the confidentiality of the credit card number, it has two problems.

1. When the `processPurchase` method succeeds, the application sends back the credit card number to the user via an HTTP response. Although the credit card number belongs to the user, it should not be sent to the user unless necessary since the number may be peeped at over the user's shoulder, or leaked from the browser cache.
2. When the `processPurchase` method fails, the application outputs the user's credit card number into a log file. Confidential information should not be output into log files unless necessary.

We propose a system which prevents such undesirable information flow. That is, in the above example, the system detects undesirable information flow and stops processing when a credit card number is being output. In addition, when data is properly sanitized (e.g., when a credit card number is masked), the data should be "declassified" to indicate that it is no longer confidential.

In order to achieve fine-grained information flow control, language-based information flow control is receiving attention [1]. Much of the past research focuses on static analysis of information flow in a program using the type system or data flow analysis. However, we consider the practical use of such technologies to be difficult,

because 1) It is difficult to analyze complicated data structures and control flows in multi-threaded object-oriented languages, 2) When a programing language is extended to include security functionality, existing software resources, such as libraries and development tools cannot be reused without adaptations, and 3) static analysis cannot take the dynamically generated code into account.

A dynamic information flow control approach has been proposed by Haldar, Chandra and Franz [2] [3] [4] to take advantage of the rich state information from a running application. They use a bytecode rewriting technique to modify the application bytecode to insert extra code for tracking the information flow of an application. Their approach tracks information propagation through method invocations and field access. We propose a Dynamic Information Flow Control Architecture for Java (DIFCA-J), inspired by the Haldar et al. approach. DIFCA-J inherits characteristics of being able to support dynamic conditions of running applications, not requiring any sourcecode from the target software, and being independent of Java VM implementations. In addition, in our system: 1) information flow is tracked and controlled at the granularity of primitive data types through most of the JVM instructions including logical and arithmetic computations, the operand stack and local variables operations, method invocation, and exceptions. 2) it effectively labels data for input or output from or to external environments, especially data exchanged with databases, and 3) it supports fine-grained application-level policies, including declassification policies. A more comparison with Haldar's approach is discussed in Section 6.

We implemented DIFCA-J on top of Apache Tomcat, and integrated Application Privacy Monitoring for JDBC (APM4JDBC) to enforce security policies in database (See Section 4).

The rest of the paper is organized as follows. Section 2 gives an overview of DIFCA-J and its basic concepts. Section 3 shows the architecture of DIFCA-J, and its detailed method for information flow control in execution of Java bytecode instructions. Section 4 describes integration of the databases for fine- grained information flow control between the database and the application server. Section 5 describes the current prototype implementation. Section 6 reviews related work, and Section 7 concludes the paper and covers our future research agenda.

2 Overview of DIFCA-J

We propose a Dynamic Information Flow Control Architecture for Java (DIFCA-J) to control the information flow of Java applications.

DIFCA-J allows administrators to define security labels for external resources (such as files, network and databases) as well as the information-flow policies between labels. During the execution of applications, DIFCA-J keeps track of propagation of security labels for the data, and detects any output that violates the policies.

The run-time functionality is inserted into the application bytecode as inline reference monitors (IRM) [5], using the bytecode rewriting technique. When the application is executed, the inserted IRM code communicates with the Access

Control Module (ACM), to notify it of the state of the running application. The ACM is implemented as a Java class, and a unique instance is created and associated with each thread. An ACM has an internal structure similar to a Java Virtual Machine (JVM). However, instead of holding data, an ACM holds security labels that are associated with data in the JVM. The ACM propagates security labels of data by synchronizing itself with the code execution of the JVM.

The ACM takes two sets of policies: the labeling policies, and the information-flow policies. Since the instrumented bytecode is independent of the policies, the policies can be late-bound to the application at run-time.

Labeling Policies. In DIFCA-J, any input or output to external resources is identified by Java APIs associated with the operation, that is represented in a form of pseudo-URI such as "java:*class_name.method_name*". Some resources require finer granular control for labels than APIs; e.g., the labels of files and network resources need to be identified by their locations, rather than by APIs. Therefore, such location are represented in the form of a URL.

In addition, a resource is either structured or unstructured, with regard to the labeling of its information. Examples of unstrucutred data are plain text files, where each file includes information with the same confidentiality. A database system is an example of a structured resource. Each datum in a database needs a more dedicated labeling policy, since each table may include columns with different classifications, and the classification may depend on the context of the query. DIFCA-J leverages Application Privacy Monitoring for JDBC (APM4JDBC) [6] to allow dynamic labeling of the database query results.

Information Flow Policies. DIFCA-J is policy agnostic and thus can flexibly adopt different types of policies, such as Biba [7], Bell-LaPadula [8], or the lattice model [9].

For the sake of simplicity, in the following example we use a simple Bell-LaPadula-type policy which has only two labels `HIGH` and `LOW`; i.e., information with the label `HIGH` cannot flow into `LOW`, while information with the label `LOW` can flow into `HIGH`. Labels are propagated when information flow occurs from explicitly labeled data.

Label Composition. When a value is derived from the composition of two values with different labels, the label of the resulting value becomes the composition of the two labels of the original values. Here, the composed label should be the Least Upper Bound (LUB) of the two labels; i.e., the lowest label which satisfy both of the two labels. For example, when $a + b = c$ where a is `HIGH` and b is `LOW`, the label of c needs to be `HIGH`. This is because when an attacker learns the values of both b and c, he can easily infer the value of a.

3 DIFCA-J Architecture

Figure 3 shows the architecture of DIFCA-J. IRMWriter takes the bytecode of an application as input, and inserts inline reference monitor (IRM) code into it.

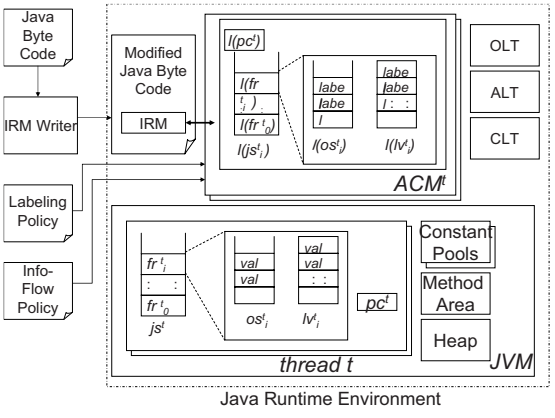


Fig. 3. DIFCA-J Architecture

This instrumentation process may happen either before application deployment, or on-the-fly when the code is loaded by a class loader.

A JVM has a JVM stack js^t for each thread t, which is a stack of frames, where each frame (fr_i^t, where i denotes the position on js^t) holds an operand stack os_i^t and a list of local variables lv_i^t for the method m_i^t that is being executed. When a new method m_{i+1}^t is called, a new frame fr_{i+1}^t is created and pushed onto js^t. Likewise, an ACM has a stack structure $l(js_t)$ corresponding to the JVM stack for each thread. Each $l(js_t)$ is a stack of frames-for-labels $l(fr_i^t)$. Instead of holding data operated upon by a method, each $l(fr_i^t)$ holds the labels of local variables $l(lv_i^t)$ and the operand stack $l(os_i^t)$ associated with the data.

As each bytecode instruction is executed in the application, the IRM code synchronizes the state of the ACM with the state of JVM so that the labels in $l(js^t)$ represents the security label of the data in js^t being operated upon in the application.

In a JVM, object instances and static field data are stored in the heap area. The ACM has three tables for holding the labels for the objects in the heap: the object label table (OLT) for the labels of objects and the fields of the objects, the array label table (ALT) for the labels of array elements, and the class label table (CLT) for the static fields of the classes.

A JVM also has a method area and a constant area for holding the bytecode of methods as well as the constant data of Java classes and interfaces. We regard these areas as non-confidential, and associate them with the special label `NONE` by default (i.e., $LUB(\texttt{NONE}, l) = l$ for any given label l, and either `HIGH` or `LOW` can flow to destination with the label `NONE`).

3.1 Information Flow in Java Bytecode

Programs written in the Java language are compiled into Java bytecode, a standard pseudo-machine language that is executed on a JVM [12]. Since we attempt

to support applications with no source code, we target the Java bytecode to track the information flow of an application.

The JVM Specification [12] defines about 200 instructions, but in many cases a single semantic operation is defined in multiple instructions for different data types. Similarly, most instructions that concern local variables have variants that include frequently used local variable indices

3.2 Stack and Local Variable Operations

When information is exchanged between the operand stack os_i^t and the local variable table lv_i^t, e.g., by a `LOAD` or `STORE` instruction, the ACM propagates the label between $l(os_i^t)$ and $l(lv_i^t)$. When two values are combined to create a new value, e.g., as a result of a binary operation, the ACM obtains the composition of the two labels and associates it with the new value.

When a constant value is loaded onto os_i^t, the `NONE` label is associated with the value (unless the load operation was affected by an implicit flow as discussed later in Section 3.5).

For example, Figure 4(a) is a simple Java program which adds the variable b and the constant 1 and assign the result into the variable a. Figure 4(b) shows the bytecode representation of the same program.

The ACM follows the operations of the JVM. For example, the ACM first loads the labels of b from $l(lv_i^t)$ to $l(os_i^t)$ at the `LOAD` operation and then the label of 1 (`NONE`) to $l(os_i^t)$. When the `ADD` operation is executed, ACM obtains the composition of the two labels on $l(os_i^t)$, in this case $newlabel = LUB(l(os_i^t[j-1]), l(os_i^t)[j])$, where j denotes the highest position of $l(os_i^t)$ as of the time the operation is performed. Then the resulting *newlabel* is pushed onto $l(os_i^t)$. Finally, the *newlabel* is propagated to $l(lv_i^t)$ to be associated with the variable a at the `ISTORE_1` operation.

3.3 Object and Field Access

Each object can be associated with a security label, but each field of an object may have different labels than the object itself. Therefore, our approach is designed to handle them separately.

A JVM stores objects into the heap area, and accesses to the fields are done through the `GETFIELD` and `PUTFIELD` instructions. The ACM stores the labels of the objects and their fields in the Object Label Table (OLT). The labels are propagated between OLT and os_i^t at each `GETFIELD` or `PUTFIELD` instruction. Similarly, the labels of static fields and the labels of array elements are managed in the Class Label Table (CLT) and the Array Label Table (ALT), respectively.

3.4 Method Invocation

When a method is invoked, information is propagated between the caller method and the callee method through method arguments and the return value.

When a method `foo()` invokes the method `bar()`, the method arguments are the output and the return value is the input, from the view point of `foo()`. In contrast, for `bar()`, the method arguments are the input and the return value is

the output. Therefore, DIFCA-J allows defining the labeling policy of input and output for `foo()` and `bar()` separately. When an explicit policy is defined for an input to a method, then a security label is associated with the input data. When an explicit labeling policy is defined for the method output, then the label of the output data is compared with the label of the method, and execution is stopped when any violation of the information flow policy is detected. For example, when some data with label `HIGH` is being specified as an argument for a method with the output label `LOW`, that invocation causes an information-flow violation.

It should be noted that both the caller `foo()` and the callee `bar()` are not necessarily instrumented with IRM. The IRM Writer can only instrument the application code, and the standard Java libraries and the middleware (e.g., Web server) must not be modified. If the standard libraries are also IRM-enabled, the IRM code would recursively call the IRM code and the code could not be executed properly. Therefore, when `foo()` invokes `bar()`, there are four possibilities with regards to their IRM enablement: 1) both of `foo()` and `bar()` are IRM enabled, 2) only `foo()` is IRM enabled, 3) only `bar()` is IRM enabled, or 4) neither `foo()` nor `bar()` is IRM enabled.

In case 1), the label of the arguments is explicitly propagated by IRM by copying the labels from caller method's $l(os_i^t)$ to the callee method's $l(lv_{i+1}^t)$, while the arguments are copied from the os_i^t to lv_{i+1}^t. When returning from the method by the `RETURN` instruction, the label of the return value is popped from $l(os_{i+1}^t)$ and pushed to $l(os_i^t)$.

In case 2), since only m_i^t is IRM-enabled, the `INVOKE` instruction is IRM-enabled, but the `RETURN` instruction that is executed from m_{i+1}^t is not IRM-enabled. Therefore, the caller method m_i^t infers the label of the return value from the composition of the labels of the input arguments and the target object itself.

In case 3), since the caller method is not IRM-enabled, the callee does not receive the label of the input values unless an explicit policy is specified and thus the label `NONE` is associated with the input argument by default.

3.5 Implicit Flow

Even when no explicit flow occurs from `HIGH` to `LOW`, the `HIGH` information can be infered from the `LOW` information when the `HIGH` information affects the control flow of the program. For example, in the example code in Figure 5(a), the value of x can be inferred from the value of y after the `if` statement, since the value of y differs by the value of x. Such information flow is called an implicit flow [29].

In order to detect such an implicit flow, DIFCA-J associates a security label with the program counter to show any implicit flow caused by the execution of the code. Let pc^t be the program counter of the thread t and $l(pc^t)$ be the label associated with pc^t. In the above example, when the value of x causes a conditional branch in Line 3, $l(pc^t)$ is raised to `HIGH`. Then when the value is assigned to y, the label `HIGH` is propagated to the label of y.

Figure 5(b) shows the Java bytecode that is compiled from the Java source code in Figure 5(a). In Line 6 in the bytecode, `IF_ICMPNE` evaluates the value on

```
1: int a = 1;
2: int b = 2;
3: a = b + 1;
```

(a) Java Program

```
0: ICONST_1
1: ISTORE_1
2: ICONST_2
3: ISTORE_2
4: ILOAD_2
5: ICONST_1
6: IADD
7: ISTORE_1
```

(b) Bytecode

Fig. 4. Code Example for Addition

```
1: x = 1; // HIGH
2: y = 0; // LOW
3: if(x == 1){
4:   y = 1;
5: }
```

(a) Java Program

```
0: ICONST_1
1: ISTORE_1 // x
2: ICONST_0
3: ISTORE_2 // y
4: ILOAD_1
5: ICONST_1
6: IF_ICMPNE #11
9: ICONST_1
10: ISTORE_2 // y=1
11: ...
```

(b) Bytecode

Fig. 5. Code Example for Branch

the os_i^t to branch conditionally. Lines 9 and 10 are the code that are executed conditionally depending on the two values that are on os_i^t (i.e., x and the constant 1). All of the values that are affected by these conditions need to be associated with the security label of the composition of the original value and the program counter. In the above example, when `IF_ICMPNE` is executed, $l(pc^t)$ becomes the composition of the label of x and the constant 1 (i.e., `HIGH`), and then in Line 9, the value propagated by the `ISTORE` instruction will be $LUB(l(os_i^t[j]), l(pc^t))$, which is the composition of the label of the value on the operand stack $l(os_i^t)$ and the label $l(pc^t)$.

Similarly, all of the IRM operations described before, need to compose $l(pc^t)$ in addition to the label of the operands. For example, when a constant value is loaded onto OS, the label $l = LUB(\texttt{NONE}, l(pc^t))$ is actually pushed onto $l(os_i^t)$. When multiple conditional branches are nesting, $l(pc^t)$ evolves to reflect the context; i.e., each time the pc^t reaches a new branch, $l(pc^t)$ is updated to be $LUB(l(pc^t), l(c))$ where $l(c)$ denotes the label of the branch condition. $l(pc^t)$ is reset at the join point of each conditional statement.

Strictly speaking, the implicit flow occurs whether or not the body of the `if` statement is executed; that is, one can infer that the value x is not 1 when the value of y is 0. It should be noted that a purely dynamic approach, can propagate the label only when the assignment is done, when there is no knowledge of other possible execution paths [25][26].

3.6 Exceptions

An exception in a Java program is caused by either 1) the JVM (e.g, division by zero), 2) a `throw` statement in the library code, or 3) a `throw` statement in the application code. In DIFCA-J, exceptions explicitly thrown by applications will be assigned the security label that is determined from the label of the data that is set in the exception as well as $l(pc^t)$ of the code that throws the exception. When an exception is not caught, the frames are popped in the JVM stack. In order to synchronize the stack depth, the IRM inserts default exception handlers to capture the exception, to synchronize the state of the ACM, and to throw the exception immediately.

When an exception is directly thrown by the JVM, the IRM's ability to detect the label of the exception object is limited. For example, when `division by zero` is reported by java.lang.ArithmeticException, it can be inferred that the dividision operand was 0. However, since this exception is caused by the JVM itself, it is difficult for the IRM to reflect the label of the operand to the label of the exception.

3.7 Multi-threading

The types of information flowing between threads in a multi-thread program can be classified into 4 types: 1) Initialization parameters of the child thread objects that are set by the parent thread, 2) Pairs of threads communicatiing through shared global objects (e.g., singleton objects), 3) Information exchanged via external resources such as files, databases, or system properties, or 4) Covert communication channels that make use of Java's multi-thread capabilities, such as thread synchronization and interruption.

In DIFCA-J, information flowing through 1) and 2) is captured since the labels of the objects in the heap area are managed by the global tables OLT, CLT, and ALT. Information exchanges through external resources can be captured as long as all of the resources are labeled properly.

3.8 Declassification

Declassification is an important issue for the practical use of an information flow control system to mitigate the label creeping problem [29]. For example, in a system that implements the Bell-LaPadula model with two labels `HIGH` and `LOW`, all processes that can read from both of `HIGH` and `LOW` information must not have a write permission to the `LOW` information. Therefore, as information propagates, information that originally had the `LOW` label tends to be associated with the `HIGH` label. Since each process is regarded as a black-box for the operating system level information flow control, it is difficult to avoid the label-creeping problem.

Even in a language-level information flow control system, the labels of data tend to become more strict as the program is executed. In particular, this trend is significant when the label of the program counter is composed into the labels of the variables to capture the implicit flows.

However in reality, not all information produced from confidential information is confidential. For example, credit card numbers are usually regarded as confidential, but it is a common practice to mask the credit card number except for the last 4 digits to make it public information that can be printed on the bill, e.g., `"****-****-****-1234"`. Another example is a password. The password itself is confidential but its hash value or a boolean result of authentication, both derived from the password, are public information.

DIFCA-J supports declassification through a API-level declassification policy specification. For example, if there is an API method `String mask(String creditcard)` that masks a credit number except for the last 4 digits, the labeling policy can be defined to force the return value of this method to `LOW`, and thus allows declassification without modifying the code.

4 Labeling on Database Queries

JDBC (Java DataBase Connectivity) is a standard API for accessing databases from Java applications. The standard API encapsulates the complexity of each database management system and its driver, and allows Java applications to utilize databases without concerns about the implementation- specific differences. A typical Java application connects to a database using JDBC, "logs-in" with a user account registered with the database, and issues queries using SQL.

When the database management system employs an access control mechanism, the access permission is linked to the given database user ID. However, many of today's Web applications use a single database account for processing requests from multiple Web users, in order to effectively reuse database connections through connection pooling to optimize performance. Therefore, the access control at the database access point is ignored.

Application Privacy Monitoring for JDBC (APM4JDBC) [6] is a generic framework for a JDBC to intercept JDBC API calls and to insert customized behavior for each activation of the JDBC. Such behavior includes application-level access control for database queries as well as for recording database accesses for auditing purposes.

Figure 6 shows the architecture of APM4JDBC. Any query context (such as a web user account that is different from the database user account) can be corrected by the Context Handler, so that such information can be later utilized by the plug-in access controllers to filter and modify the SQL queries and responses.

DIFCA-J uses the APM4JDBC framework to collect the contexts of the database queries, and labels the retrieved data based on the policies and the contexts of the queries. For example, DIFCA-J allows putting different security labels for each column on the queried data (e.g., associate the `HIGH` label with the credit card number in our example scenario introduced in Section 1). Similarly, DIFCA-J allows controlling data output into the database (e.g., make sure that application will not write a value with the `HIGH` label into a database column which should only hold the `LOW` values). In addition, we can extend the labeling system to a richer model, such as the lattice model, and associate different labels for each user's transactions, and prevent contamination of information belonging to different users.

5 Prototype Implementation

DIFCA-J was prototyped on top of the Apache Tomcat Web container with JVM 1.5. The IRM Writer was implemented with Apache Byte Code Engineering Library (BCEL) [14], an open source toolkit for analyzing and modifying arbitrary Java bytecode. We modified the Web Application class loader in Tomcat to instrument only the bytecode of application classes when they are being loaded. The ACM is implemented as a singleton Java object.

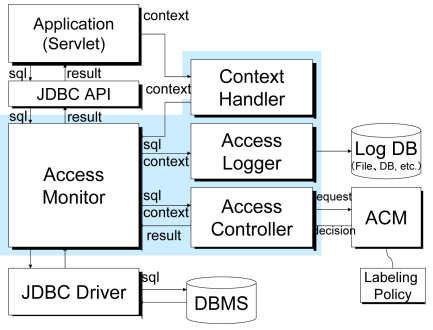


Fig. 6. APM4JDBC

```
...
iconst_0
invokestatic acm.ACM.loadSingleVar (I)V
aload_0
invokestatic acm.ACM.pushSingleConst ()V
iconst_0
dup2
invokestatic acm.ACM.loadSingleArray c
aaload
iconst_1
invokestatic acm.ACM.storeSingleVar (I)V
astore_1
iconst_0
invokestatic acm.ACM.loadSingleVar (I)V
aload_0
invokestatic acm.ACM.pushSingleConst ()V
...
```

Fig. 7. IRM Enabled Bytecode

Figure 7 shows an example of IRM-enabled bytecode after instrumentation. Lines with underlines are inserted IRM code that calls the ACM by invoking the methods of the ACM.

Limitations of the current prototype are that it does not support exceptions, and some policies (i.e., information flow policy and the database policies) are hard wired in Java code.

5.1 Policy Definition

The sample information flow policy is defined in a Java class which provides label comparison and composition as methods. The sample labeling policy (Fig. 8) defines the label `LOW` for the HTTP requests and responses, and to the `printlog()` method. The policy on the `mask()` method defines the declassification policy on the masked credit card numbers.

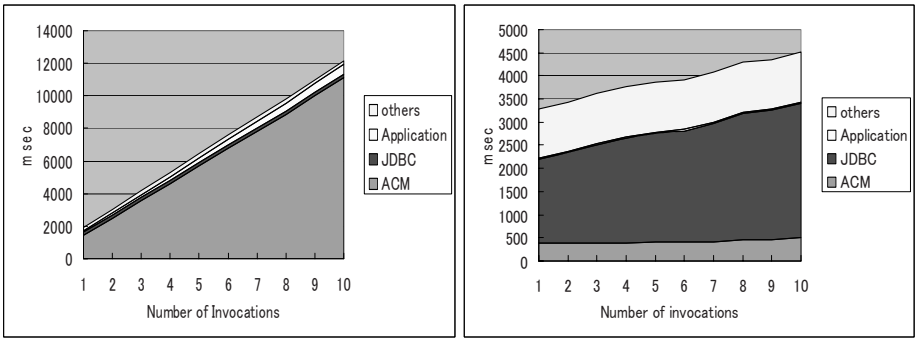
DIFCA-J requires labeling policy only on API that concern input and output of data. Therefore, the administrator's burden of policy definition is smaller than security enhanced language such as Jif [10]. E.g., only 4 entries of labeling policy is required in example application in Section 1; other methods that does not concern with input and output of data will just propagate the label.

When no explicit policy is defined, DIFCA-J infers that a label of the value returned from a standard library method is the composition of the labels of the target object and the arguments. When this inference rule fails, explicit labeling policies need to be defined for such API. However, it is possible to pre-define a set of policies for each standard library and deploy with DIFCA-J, in order to mitigate administrator's burden to define them by themselves.

The policy on the database, which associates the label `HIGH` with credit card numbers, is hard-coded in a Java class in the current prototype. But it is obvious that we can extend the system to allow more flexible policy definition. Since the example policy adopts the simplest two-level labels, no context information was used for labeling the database query results. However, the architecture is policy agnostic and we can easily extend the policy to accommodate finer-grained

```
<Policy>
 <InputRule><Label>LOW</Label><Type>argument</Type>
   <URI>java:foo.shop.Purchase.doGet</URI></InputRule>
 <InputRule> <Label>LOW</Label><Type>return</Type>
   <URI>java:foo.shop.Purchase.mask</URI></InputRule>
 <OutputRule><Label>LOW</Label><Type>argument</Type>
   <URI>java:foo.shop.Purchase.printlog</URI></OutputRule>
</Policy>
```

Fig. 8. Example Policy



(a) Arithmetic Operations Web App. (b) JDBC Web App.

Fig. 9. Performance Evaluation

policies in more flexible way; e.g., the credit card numbers that belong to different users are associated with different labels that corresponds to the user identities.

5.2 Performance Evaluation

We measured performance overhead of DIFCA-J in two types of Web applications, 1) only arithmetic operations without JDBC (Fig. 9(a)), and 2) sample application in Section 1 which uses JDBC iFig. 9(b)j. We ran each application for 10 times, and measured accumulated time consumption of each Java package using a profiler. The Web server (Apache Tomcat) and the MySQL database were set up on the same computer. In 1), the pure overhead by ACM over the original application class is about 18-24 times. In 2), JDBC connection and query consumes more than 90% of the entire execution time (even if excluding the connection establishment which occurs only once), and thus the overhead by ACM stays around 10%. But the pure overhead is about 13-18 times. IRMWriter instruments the bytecode only once when the application is loaded, and thus most of the overhead is resulted from ACM. Note that the pure overhead means the execution time comparison between the original application class itself and ACM, and the overhead in the turn-around-time of the Web application as a whole is smaller, especially when JDBC is involved. After instrumentation, each class file increases its size about 2-2.8 times.

In conclusion, although performance overhead caused by ACM is not small, we think that it is still acceptable in typical Web applications which uses JDBC.

6 Related Work

A number of prior research [7][8][9] have studied policy models to guarantee secure information flow. There are also technologies to implement the models. For example, the multi-level security (MLS) system implements the Bell-LaPadula [8] model, providing strong isolation from the hardware and the networks.

When the information flow control is implemented at the granularity of a system or a process, it is inevitable that securely designed information flow policies cause the label creeping problem [29], since the classification of information becomes more strict thorough propagation, making it increasingly difficult for the information to be shared effectively. Language-based information flow control is getting attention in order to allow for fine-grained information flow control. This section briefly reviews prior research work. The language-based information flow control can be classified into static approaches and dynamic approaches. More thorough survey of static approachs is found in [1].

Static Approaches. Kobayashi and Shirane [15] defined a small subset of Java bytecode and statically analyzes information flow within it. Barthe et al. [16][27][17] proved noninterference for a subset of Java bytecode and proved that the program written in a security-enhanced language can be compiled into bytecode without weakening the security properties. No implementation was reported. Genaim and Spoto [18] studied the information flow analysis of a more complete set of Java bytecode, and implemented their proposed method. Yu and Isam proposes Typed Assembly Language for Confidentiality (TALc) [28] for information flow analysis and proved its noninterference.

Jif [10][19][20] is an extension of the Java language, which allows defining security labels as types of program constructs. Programs written in Jif are compiled into ordinary Java bytecode, and thus have no dependencies on the run-time environment. However, existing applications need to be converted to Jif programs. Recently, Boniface et al. implemented an e-mail application in Jif [21] and evaluated its usefulness in realistic applications. Li and Zhancewic [22] addressed the information flow problem in Web applications with the security-enhanced scripting language. Static approaches for the declassification problem are found in [23] [13].

Dynamic Approaches. A dynamic approach is potentially more precise than the static approach since t can exploit the detailed conditions of the running programs. It also allows security policies to be defined dynamically.

Beres and Dalton [11] modified the operating system, to track information flow in the execution of machine language. A similar approach is applicable to Java, but it requires modification of the Java VM, and the resulting implementation is dependent on the JVM.

Erlingsson et al. [5] proposed inserting Inline Reference Monitors (IRMs) into Java bytecode to implement access control that is equivalent to the Java2 security architecture [24]. Haldar, Chandra, and Franz [2] [3] [4] proposed information flow control for Java using a bytecode rewriting technique. Their initial focus included "taint" propagation for values input into web applications, but can be easily extended to support richer labeling systems. According to Franz [4], they support label propagation at the granularity of Java objects and fields. The dynamic mechanism tracks information propagation at the time of method invocations and field access. They also employ static analysis for detecting implicit flows. The literature does not say if that their approach supports information propagation through all JVM instructions that involve operand stacks and local variables.

DIFCA-J was influenced by [2] [3] [4], but the major difference is the granularity of the label propagation. DIFCA-J supports information propagation through most of the JVM instructions, including arithmetic operations, array elements, multi-threading, and exceptions, and a policy based declassification mechanism. We also integrated APM4JDBC to effectively label database query results and to control input and output to the database through JDBC.

The problem of implicit flow in dynamic approach is addressed in [25][26] based on combination of dynamic and static analysis. Shroff [26] also addresses the problem by analyzing dependencies of data through monitoring multiple executions of the program. Since our work aims at detecting undesirable information flow by programming errors without modifying the application source code, it is not the focus of this paper to detect all implicit flows. However, we believe that the technique presented in this paper can be extended to collect information about the data dependencies and to detect indirect implicit flows.

7 Conclusions and Future Work

This paper proposes DIFCA-J, which enforces the language-based information flow control policies for Java applications. We use a bytecode rewriting technique to insert inline reference monitors (IRMs), and thus 1) the IRMs can utilize the detailed conditions of the running applications, 2) it does not require source code of the target application, and 3) the system is independent of JVM implementations. DIFCA-J tracks the propagation of information in the program through most of the JVM instructions, and controls the input and output to the external environment based on the given information flow policies. DIFCA-J also intercepts the JDBC queries to effectively label the query results, and control input to and output from the database.

However, the current proposal still leaves gaps for future research. First, the purely dynamic approach can discover only information flows that are actually executed, and especially cannot detect all of the implicit flows. Second, the current approach requires a bytecode-level IRM to be inserted for every bytecode instruction of the original code, and causes significant performance overhead. Third, since *OLT* stores object references with associated security labels, it

prevents target objects from garbage collected, and causes memory overhead at run-time. Fourth, terminating the transaction due to the information flow violation may cause problems in database consistency. It is inherently difficult to handle such exceptions without modifying the applications. Some of the problems may be acceptable when using DIFCA-J for the pre-deployment test, but care needs to be taken to define the test cases with good coverage.

Usability and policy specification is another challenge that needs to be addressed. DIFCA-J does not require the source code of the target applications, but the policy writer still needs to understand the structure of the program and the semantics of the methods. Especially when a declassification policy is defined for a method, such a definition may easily introduce human error, unless the semantics of the method are well defined and the consequences of the declassification are well understood. This is a future topic for allowing easy and safe policy definitions without needing knowledge of the source code.

Acknowledgement

The authors wish to thank anonymous reviewers as well as our colleagues at the IBM Tokyo Research Laboratory and Institute of Information Security for their feedback and insights on earlier versions of this paper. This study was partly sponsored by the Ministry of Economy, Trade and Industry, Japan (METI) under a contract for the New-Generation Information Security R&D Program.

References

1. Sabelfeld, A., Myers, A.C.: Language-Based Information Flow Security. IEEE Journal on Selected Areas in Communications 21(1) (2003)
2. Haldar, V., Chandra, D., Franz, M.: Dynamic Taint Propagation for Java. In: Srikanthan, T., Xue, J., Chang, C.-H. (eds.) ACSAC 2005. LNCS, vol. 3740, Springer, Heidelberg (2005)
3. Haldar, V., Chandra, D., Franz, M.: Practical, Dynamic Information Flow for Virtual Machines. In: PLID (2005)
4. Franz, M.: Moving Trust Out of Application Programs: A Software Architecture Based on Multi-Level Security Virtual Machines (TR. 06-10), UC Irvine (2006)
5. Erlingsson, U., Schneider, F.B.: IRM Enforcement of Java Stack Inspection. In: IEEE Sympo. on S&P, IEEE Computer Society Press, Los Alamitos (2000)
6. Application Privacy Monitoring for JDBC (APM4JDBC): IBM AlphaWorks
7. Biba, K.: Integrity Considerations for Secure Computer Systems (MTR-3153). Technical report, MITRE (1975)
8. Bell, D.E., LaPadula, L.J.: Secure Computer System: Unified Exposition and Multics Interpretation (MTR-2997 Rev. 1). Technical report, MITRE (1976)
9. Denning, D.E.: The lattice model of secure information flow. Communications of the ACM 19(5), 236–243 (1976)
10. Myers, A.C.: JFlow: Practical Mostly-Static Information Flow Control. In: POPL (1999)
11. Beres, Y., Dalton, C.: Dynamic Label Binding at Run-time. In: New Security Paradigms Workshop (NSPW) (2003)

12. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley, Reading (1999)
13. Li, P., Zdancewic, S.: Downgrading policies and relaxed noninterference. In: POPL'05. Symposium on Principles of Programming Languages (2005)
14. Apache Byte Code Engineering Library (BCEL), `http://jakarta.apache.org/bcel/`
15. Kobayashi, N., Shirane, K.: Type-based Information Flow Analysis for Low-Level Languages. In: APLAS 2002 (2002)
16. Barthe, G., Basu, A., Rezk, T.: Security Types Preserving Compilation. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, Springer, Heidelberg (2004)
17. Barthe, G., Naumann, D.A., Rezk, T.: Deriving an Information Flow Checker and Certifying Compiler for Java. In: IEEE Sympo. on S&P, IEEE Computer Society Press, Los Alamitos (2006)
18. Genaim, S., Spoto, F.: Information Flow Analysis for Java Bytecode. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, Springer, Heidelberg (2005)
19. Zdancewic, S., et al.: Untrusted Hosts and Confidentiality: Secure Program Partitioning. In: SOSP. Symposium on Operating Systems Principles (2001)
20. Zheng, L., Chong, S., Myers, A.C., Zdancewic, S.: Using replication and partitioning to build secure distributed systems. In: IEEE Sympo. on S&P, IEEE Computer Society Press, Los Alamitos (2003)
21. Hicks, B., et al.: From Languages to Systems: Understanding Practical Application Development in Security-typed Languages. In: Jesshope, C., Egan, C. (eds.) ACSAC 2006. LNCS, vol. 4186, Springer, Heidelberg (2006)
22. Li, P., Zdancewic, S.: Practical Information-flow Control in Web-based Information Systems. In: CSFW (2005)
23. Myers, A.C., Sabelfeld, A.: Enforcing Robust Declassification. In: CSFW (2004)
24. Gong, L., et al.: Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2, USITS (1997)
25. Guernic, G.L., et al.: Automata-based Confidentiality Monitoring. In: ASIAN'06. Annual Asian Computing Science Conference (2006)
26. Shroff, P., Smith, S.F., Thober, M.: Dynamic Dependency Monitoring to Secure Information Flow. In: IEEE Computer Security Foundations Symposium, IEEE Computer Society Press, Los Alamitos (2007)
27. Barthe, G., Rezk, T.: Non-interference for a JVM-like language. In: TLDI (2005)
28. Yu, D., Islam, N.: A Typed Assembly Language for Confidentiality. In: Sestoft, P. (ed.) ESOP 2006 and ETAPS 2006. LNCS, vol. 3924, Springer, Heidelberg (2006)
29. Denning, D.E.: Cryptography and Data Security. Addison-Wesley, Reading (1982)