Proceedings of the International MultiConference of Engineers and Computer Scientists 2012 Vol I,
IMECS 2012, March 14 - 16, 2012, Hong Kong

Design and Implementation of DF-Salvia which Provides Mandatory Access Control based on Data Flow

Shozo Ida, Takehiro Kashiyama, Eiji Takimoto, Shoichi Saito, Eric Wallace Cooper, and Koichi Mouri

***Abstract*—Recently, incidents in which data such as private information has leaked have occurred frequently. In many cases, the main causes of data leakage are as follows: taking data out illegally or unfairly, erroneous operation by a user with authority to access the data. We developed the operating system Salvia for the purpose of preventing data leakage resulting from these causes. Salvia provides the capability to attach data protection policies to each file. In addition, Salvia monitors resource access that may incur the possibility of data leakage. When a process requests to access to such resources, Salvia allows the operation only if it does not violate the policies of all files which are read by the process. That is, Salvia controls resource access by process. In this paper, we propose DF-Salvia, based on Salvia. An access control unit of DF-Salvia is data flow, which is finer-grained than the process-based access control of Salvia. This means that DF-Salvia applies a policy not to each process but to each data flow in a process in order to limit the extent of the effect of the policy to corresponding data flow. The results show a solution to the problem of over-restriction of irrelevant data.**

***Index Terms*—file access control, secure OS, data flow analysis.**

I. Introduction

RECENTLY with the popularization of computer systems, digitization of private information has been promoted. As a result, working efficiency and quality of service have been improved. On the other hand, the digitization has caused data leak incidents through networks and removable storage media. The data leak incidents result in invasions of privacy. In addition, leaks worsen a corporate image and impose monetary burdens on corporations.

In response to this situation, the Personal Information Protection Law [1] went into effect on April 1. 2005, which establishes duties and constraints on handling of personal information by companies and local governments. As a result, awareness of the importance of personal information protection is increasing but data leakage incidents still occur. The literature [2] on surveys of private information leakage incidents has reported that many data leak incidents are caused by erroneous operation, theft, mismanagement, and loss or misplacement. As an example of erroneous operation, cases of sending mail to which private information is attached have been reported. As an example of theft, mismanagement, and loss or misplacement, cases of taking out storage media on which private data of customers is stored have been reported one after another.

Some data leak incidents are caused by users with authority to data access rather than illegal access. The security mechanisms which are designed to prevent external attacks, such as encryption technology or authentication technology cannot prevent data leaks caused by such users. In the above situations, data protection mechanisms which prevent private data leaks should be developed. In addition, the mechanism should consider the following.

- Private Data Protection
  After a consensus is built on the handling conditions of private data between the owner and receiver, the private data is provided from owner to receiver. Handling conditions of private data differ depending on the owner. Therefore, the data protection mechanism should achieve access control which reflects the objectives of the owner.
- Adaptation to existing applications
  Private data is used by various applications. If a security configuration for each application is required, that raises the probability of erroneous operation. Therefore, the data protection mechanism should have the ability to adapt to existing applications.
- Data protection considering usability
  The data protection mechanism has the possibility to be used by various companies or local governments. Introduction of the mechanism must not cause lower efficiency of work for example cases in which available applications are limited or authentication is required every time private data is used.

We developed the operating system Salvia [3] which provides a data protection mechanism that satisfies the above criteria. Salvia provides the capability to configure data protection policy and attach a data protection policy to each file. In addition, Salvia enforces mandatory access control on processes that read data from protected files. More specifically, when a process seeks to access a resource which has a possibility of data leakage, Salvia allows the access only if the access does not violate any policy which was read by the process. That is, Salvia controls resource access by processes. Salvia can adapt the data protection mechanism to all applications executed on the operating

Manuscript received December 30, 2011; revised January 18, 2012.

S. Ida is with Department of Computer Science, Ritsumeikan University, Shiga, JAPAN e-mail: sida@asl.cs.ritsumei.ac.jp

T. Kashiyama is with Ritsumeikan Global Innovation Research Organization, Ritsumeikan University, Shiga, JAPAN e-mail: t-kashi@fc.ritsumei.ac.jp

E. Takimoto is with Department of Computer Science, Ritsumeikan University, Shiga, JAPAN e-mail: takimoto@asl.cs.ritsumei.ac.jp

S. Saito is with Graduate School of Engineering, Nagoya Institute of Technology, Nagoya, JAPAN e-mail: shoichi@nitech.ac.jp

E. W. Cooper is with Department of Computer Science, Ritsumeikan University, Shiga, JAPAN e-mail: cooper@is.ritsumei.ac.jp

K. Mouri is with Department of Computer Science, Ritsumeikan University, Shiga, JAPAN e-mail: mouri@cs.ritsumei.ac.jp

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

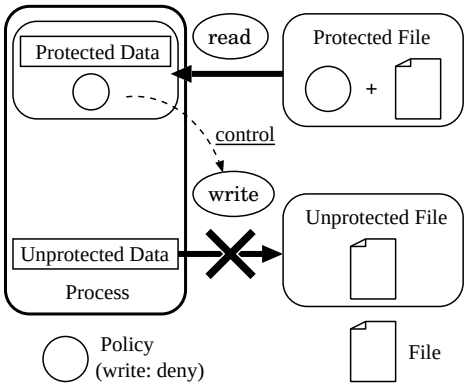

Fig. 1. Excessive Access Restriction in Salvia.

system transparently.

In this paper, we propose DF-Salvia which can control access to resources based on data flow, a finer-grained object than a process. DF-Salvia distinguishes the data processes write to resources, and determines a policy for access control according to the file on which the data had been stored. This method solves the problem of excessive access control enforcement in Salvia.

In this paper, Section 2 describes Salvia. Section 3 proposes an access control method in DF-Salvia, and Section 4 describes the implementation. Section 5 describes the evaluation and Section 6 describes related works.

II. Access control method of Salvia

A. Overview

Salvia [3] is an operating system which provides mandatory access control to prevent private information from leaking by authorized users. Because private data should be handled in keeping with the intent of its purveyor, Salvia provides the capability to attach data protection policy to each file. A data protection policy is a list of control conditions and access control configurations. A state of process or computer, which is called context, such as user ID, time, location of computer and IP address of destination terminal, is available as a condition of policy. Salvia can selectively control file access which have the possibility of data leakage based on context. For example, Salvia can achieve access control as follows: access to the protected file at between 19:00 and 7:00 is prohibited, writing data to USB flash memory is prohibited, and writing data to the network is prohibited.

Information leakage occurs in processes which read protected data. Therefore, when a process tries to read data from a protected file via system calls, Salvia hooks the system call and reads the protection policy of the file. From that point on, Salvia starts monitoring the behavior of the process and controlling its resource access. In a monitored process, resource access is restricted based on the read protection policy. When a system call for access to a resource (e.g., file and socket) is executed, Salvia determines whether to execute the system call according to the read protection policy and context.

B. Issues

Salvia enforces access control on the process which reads protected data according to the policy. More specifically, when a process attempts resource access which has a possibility of data leakage, Salvia allows the resource access only if the resource access does not violate any policy which was read by the process. That is, Salvia controls resource access based on process. The access control achieves data protection that considers privacy, but the access control has the possibility to restrict resource access excessively.

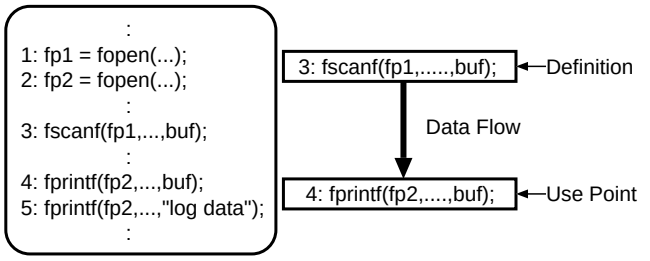

Fig. 2. Definition-Use Chain.

Fig.1 shows an example in which a process reads data from a protected file and writes the data on another file. In this case, all data writing is prohibited even if the data does not include protected data. For this reason, the following programs have the possibility to not operate as intended.

- Programs which create temporary files
  Even if a process does not attempt to write protected data to a temporary file, creation of a temporary file is prohibited.
- Programs which generate a log or configuration file
  As with programs which create a temporary file, creation of log or configuration file is prohibited.
- Programs which open more than one file at once
  In a case of an editor, if protected data has been read into an edit buffer, the editor cannot save the data on other edit buffers even if there has been no data exchange between edit buffers.

In all of the above, each program should be able to operate normally, unless there is data exchange between protected file and other files. The cause of this problem is that Salvia restricts a resource access based on a protection policy which is applied not to data but processes. Solving this problem requires access control which distinguishes each data and determines the protection policy according to the data.

III. Access Control Model of DF-Salvia

To solve the problem that access control of Salvia has the possibility to restrict resource access excessively, units of access control must be divided smaller than processes. Therefore, we propose DF-Salvia, which achieves resource access based on data flow by cooperating with a compiler. This section explains data flow analysis on a compiler and access control method of DF-Salvia.

A. Data Flow Analysis

A compiler analyzes source code and then generates the object code. At the time of compilation, a compiler performs code optimization in order to decrease the size of object code and to make its execution faster [4]. The code optimization needs to comprehend flow of variable definitions in source code. The flow of variable definitions is called data flow. A

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

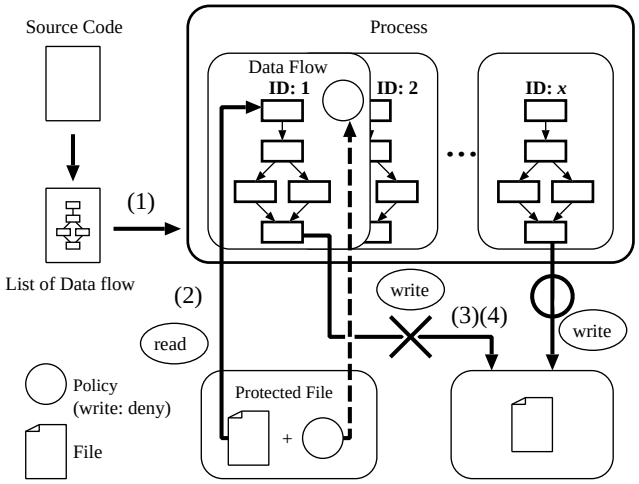

Fig. 3. Access Control Model of DF-Salvia.

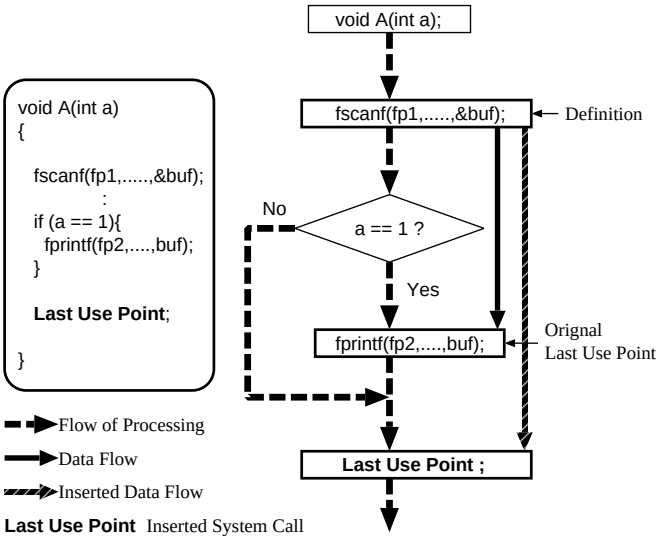

Fig. 4. Insertion of Last Use Point by Compiler.

compiler can analyze various kinds of data flows. DF-Salvia uses definition-use chain analysis which is one kind of flow. Definition-use chain consists of a definition statement which defines value of a variable and a set of use statements which use defined variables.

Fig. 2 shows the example of definition-use chain. The right side of Fig. 2 is definition-use chain of the program, which is shown in the left side of Fig. 2. Variable *buf* which is defined by function *fscanf* in the third line is used by function *fprintf* in the fourth line. More specifically, a library function or system call which reads data from a file is analyzed as a definition statement, and a library function or system call which writes data that has been read by the definition statement is analyzed as a use statement. The reason that the definition statement is limited to library functions or system calls which read the data from files is that only these statements define variables which read from protected files.

By using definition-use chains, DF-Salvia understands the statements which use variables on which protected data is stored, and determines the protection policy according to the data used. In Fig. 2, when protected data is read by function *fscanf* in the third line, DF-Salvia enforces access control only on the function in the fourth line.

B. Access Control Method

DF-Salvia changes the monitor object from process to data flow in order to achieve access control based on data flow. Fig. 3 shows the access control method of DF-Salvia. DF-Salvia performs access control with the following procedure (the numbers of procedure are also indicated in Fig. 3).

1) When a process is executed, DF-Salvia imports the data flow information which has been analyzed at the time of compilation.
2) When a read system call is invoked, the data protection policy of the file being read is applied to data flow which contains the library function call that invoked the read system call.
3) When a write system call is invoked, DF-Salvia checks the data flow, which contains the library function call that invoked the write system call.
4) If data protection policies have been applied to the data flow, DF-Salvia controls the write system call based on the data protection policies.

DF-Salvia uses the instruction address of the library function call that invoked the system call in order to identify data flows which contain library function calls that invoked the system call in the procedures 2) and 3). Specifically, DF-Salvia refers to a Data Flow ID List which is generated by data flow analysis at compilation. The Data Flow ID List has the following elements.

- The instruction address of the library function call in user program
- Data Flow ID
- The Last Use Point flag (as discussed later)

When a system call is invoked, DF-Salvia analyses the instruction address of the library function call which invoked the system call by tracing back the stack of the process. In addition, DF-Salvia identifies Data flow ID by comparing the analytical instruction address and the instruction addresses contained in the Data Flow ID List.

We are also developing an automatic Data Flow ID List generation system by reworking COINS [5], which will be described in future works.

C. Manage of Data Flow

DF-Salvia manages data protection policies based on the Data flow ID of each process because the access control unit of DF-Salvia is data flow. Specifically, when protected data is read, the protection policy is registered on the policy managent list (policy list) whose keys are process ID and Data Flow ID. DF-Salvia finds appropriate policies with these keys. Thus DF-Salvia can apply different policies to each data flow. However, it is necessary to solve two problems for the above policy management. The problems are described as follows.

1) Deletion of Policy: Salvia deletes policies which were applied to a process from the policy list, when the process terminates. On the other hand, DF-Salvia deletes the policies, when the last statement of data flow terminates. A protection policy is used only for access control of the operation of data

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

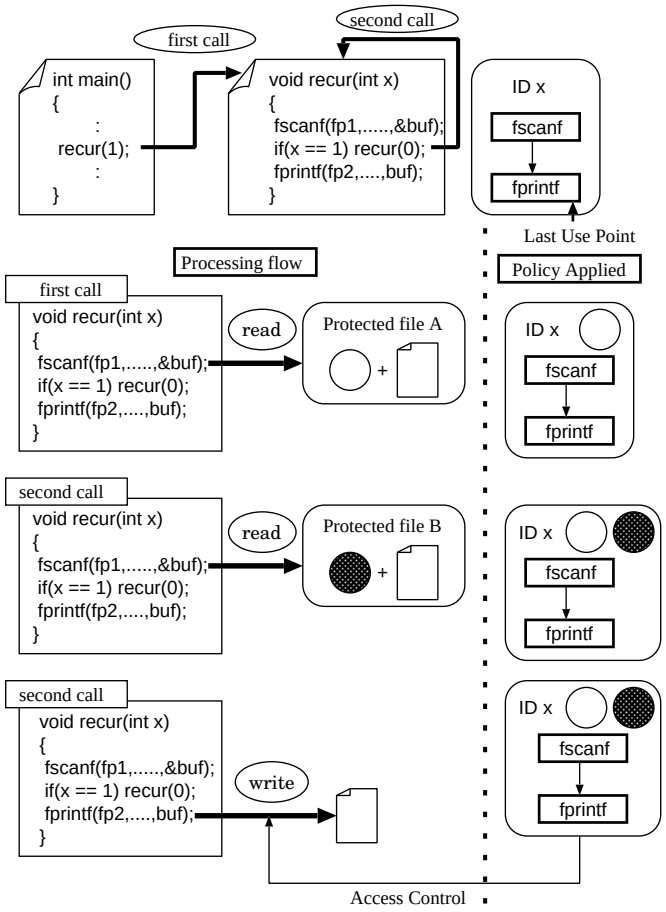

Fig. 5. Data Flow ID in Recursive Function.

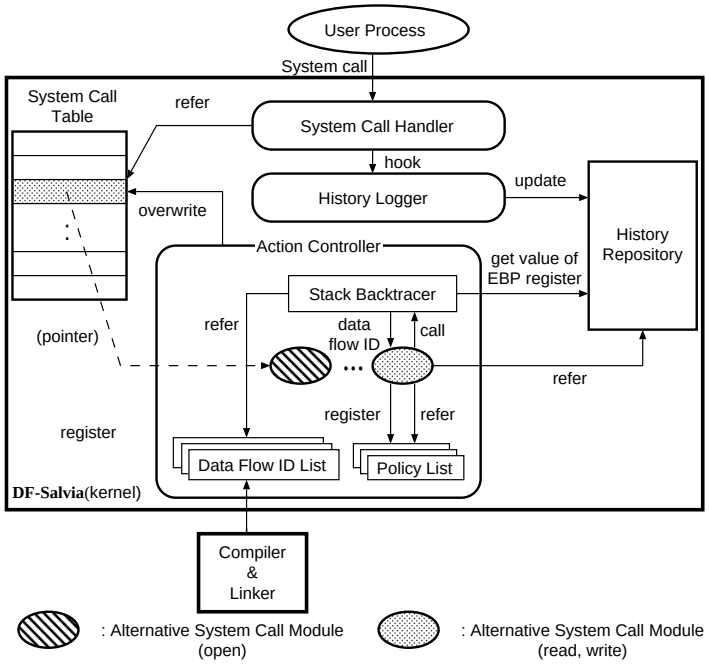

Fig. 6. System Structure of DF-Salvia.

flow to which the policy has been applied. Therefore, if the last statement of the data flow, called the Last Use Point, has terminated, the policy is unnecessary.

In summary, after access control on the Last Use Point, DF-Salvia deletes the protection policies applied to data flows which contain the Last Use Point. However, if the Last Use Point is surrounded by an *if* or *loop* statement, the above policy deletion process is not enough. If the Last Use Point is surrounded with *if* statement, the operation of Last Use Point may not execute. In this case, policies are not deleted. If the Last Use Point is surrounded by a *loop* statement, the protection policy is deleted in the first loop. Thus, the access control of Last Use Point cannot execute correctly in the second loop, because the policy to which the access control should refer does not exist. To solve this problem, the Last Use Point is set at the end of *if* statements and *loop* statements at the time of compilation.

Fig. 4 shows an example of setting the Last Use Point into end of an *if* statement. In this case, DF-Salvia can delete data protection policies which have been read on function *fscanf* because the operation of Last Use Point always eventually executes regardless of branching by *if* statements.

2) Distinction Between Policies: Fig. 5 shows the example of access control based on a recursive function. The function *main* invokes the function *recur* which has Data flow ID x twice. The function *recur* invokes itself. The point is that the definition statement operation executes once again, before the Last Use Point operation executes. Thus, if function *fscanf* reads data from different protected files in the second call and the first call, two protection policies were applied to Data flow ID x. Accordingly, although the second call of function *fprintf* is independent from the first call of function *fprintf*, the operation of function *fprintf* in the second call is controlled excessively based on the two policies.

Hence, it is necessary to distinguish data protection policies applied to data flow based on each processing of recursive function call. In order to achieve this, DF-Salvia assigns classification numbers to data flow. Specifically, when the operation of definition statement executes, DF-Salvia assigns a classification number (1 of x-1 or 2 of x-2 in Fig. 5) to the data flow. DF-Salvia applies the data protection policy to the data flow which has a classification number that corresponds to the definition statement execution. Consequently, DF-Salvia can deal with data flows of both sequential processing and iterative processing.

IV. Implementation

A. System Structure

DF-Salvia has been implemented in the Linux-2.6.8 kernel which is based on the Intel x86 architecture. Fig. 6 shows the system structure of DF-Salvia. The data protection mechanism of DF-Salvia is constructed with the following three modules; *History Logger* which gets histories of system calls, *History Repository* which stores the histories of system calls as time-series data, and *Action Controller* which controls system calls based on the contexts.

History Logger gets the time, the call's arguments, the call's return value, and the attribute value of each process when target system calls are invoked. Target system calls contain the following three types; factors of data leakage (read, etc), triggers of changing values which have possibilities to be context (setuid, etc) and triggers of DF-Salvia operation (open, etc). The attribute value of a process is

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

context, which has been stored in the process structure (user ID, etc).

History Repository manages histories of system calls which *History Logger* gets and stores as time-series data.

Action Controller reads policies and interprets them, and determines availability of system call execution. The access control of each system call needs to understand the data structure of its arguments. Therefore, *Alternative System Call Module*s are prepared for each system call. When an *Alternative System Call Module* determines availability of a system call execution, it refers to contexts which are managed by *History Repository*. *Policy List* stores policies which were read into DF-Salvia. When DF-Salvia reads the protected data, the protection policy is read, and is stored to *Policy List*. *Stack Backtracer* identifies Data Flow ID which is necessary for registering policies and control of writing data to resources. *Data Flow ID List* stores data flow information that is necessary for *Stack Backtracer* to identify Data Flow ID.

Details of *History Logger* and *History Repository* have been described in the literature[3].

B. Identify Data Flow ID

Stack Backtracer identifies Data Flow ID which contain library functions that invoke the system calls. *Stack Backtracer* has been implemented on the *Action Controller Module* by LKMs. *Stack Backtracer* is invoked by each *Alternative System Call Module* and returns the Data Flow ID, which contains a library function that invokes the *Alternative System Call Module*. The specific procedure is shown below.

1) When system calls are invoked, each *Alternative System Call Module* invokes *Stack Backtracer*.
2) *Stack Backtracer* gets the value of EBP register at the time the system call is invoked from *History Repository*.
3) *Stack Backtracer* analyzes stack of the process by using the value of the EBP register and gets the instruction address of the library function call.
4) *Stack Backtracer* gets the Data Flow ID by comparing the obtained address to the addresses which have been stored in *Data Flow ID List*, and *Stack Backtracer* returns the Data Flow ID to the *Alternative System Call Module*, which is the invoker.

The value of EBP register which is used by the procedure 3) is the value of the stack base pointer. The return address of a library function can be identified by tracing back stack frames based on the base pointer. DF-Salvia identifies the instruction address of the library function call by using its return address.

C. Procedure of Access Control

This section explains the procedure of access control on each system call.

First, the procedure for open system calls is shown below.

1) When a process opens a protected file, DF-Salvia begins to monitor the process. In addition, the *Alternative System Call Module* related to open system calls overwrites the read/write function entry in file operation table of the file object.

TABLE I
DATA FLOW ID LIST.

Line number	Instruction address	Data Flow ID	Last Use Point flag
18 fgets	0x080485d9	1	false
19 fgets	0x080485f6	2	false
21 fputs	0x0804860b	1	true
23 fputs	0x08048631	2	true

2) When the monitored process invokes a system call, *History Logger* gets the history and registers it on *History Repository*.

Next, the procedure for read system calls is shown below.

1) When a process reads protected data, *Alternative System Call Module* related to the read system call is invoked instead of read system call. the *Alternative System Call Module* reads and interprets the policy of the protected data, and determines the availability of read system call execution.
2) If the execution of read system call is allowed, *Alternative System Call Module* executes the read system call and then invokes *Stack Backtracer* to get the Data Flow ID which contains the library function that invoked the read system call. Otherwise, *Stack Backtracer* deletes the data protection policy which is read.
3) *Alternative System Call Module* registers the data protection policy on *Policy List*. The classification number is calculated from the number of same Data Flow ID, as registered in *Policy List*.

Finally, the procedure for write system calls is shown below.

1) When a process writes data on computer resources, *Alternative System Call Module* related to the write system call is invoked instead of the write system call. The *Alternative System Call Module* invokes *Stack Backtracer* to get the data ID which contains the library function which invoked the write system call. In addition, the *Alternative System Call Module* checks whether the library function is the Last Use Point or not.
2) The *Alternative System Call Module* gets the data protection policy from *Policy List* with the Data Flow ID, and then determines the availability of write system call execution based on the policy.
3) If the library function which invoked the write system call is the Last Use Point, the *Alternative System Call Module* removes the policy from *Policy List*.

V. EVALUATION

A. Functional Evaluation

We have tested to confirm whether DF-Salvia can control accessing computer resources based on data flow.

1) Experiment Description: Fig. 7 shows a test program (copy.c) and two protected files (Personal_Information_addr and Personal_Information_tel) which were used in the experiment. Table I shows a Data Flow ID List generated by the use of *objdump*. The test program reads one sentence data from each protected file protected file and writes it into another file. The policy of Personal_Information_addr

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

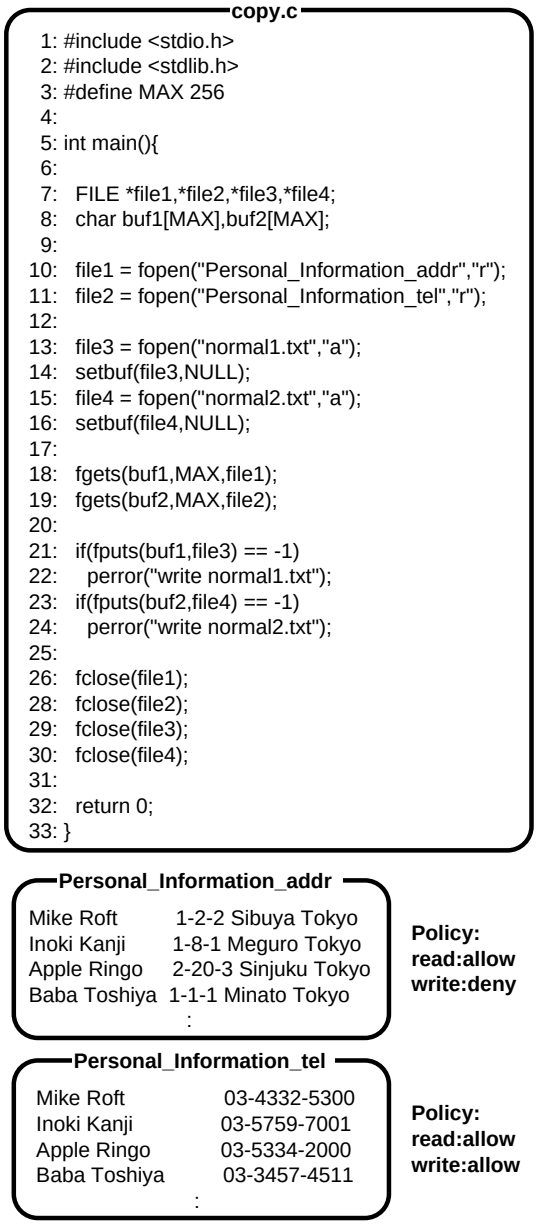

Fig. 7. Test Program and Protected Files.

denies writing its protected data. The policy of Personal_Information_tel allows writing its protected data.

The Data Flow ID List of this test program is shown in Table I. Definition statements are function *fgets* in line 18 and line 19 which read protected data from each files. Use statements are function *fputs* in line 21 and line 23 which write each data which were read in line 18 and line 19. Therefore, the test program has two data flows. Data flow ID 1 is the data flow which contains function *fgets* in line 18 and function *fputs* in line 21. Data flow ID 2 is the data flow which contains function *fgets* in line 19 and function *fputs* in line 23. In addition, function *fputs* in line 21 and line 23 are Last Use Point, because use statement which corresponds to function *fgets* in line 18 or line 19 does not exit after line 21 and line 23.

2) Experiment Result: Fig. 8 shows an execution result of the test program with Salvia. Salvia enforces access control which based on Personal_Information_addr which was read

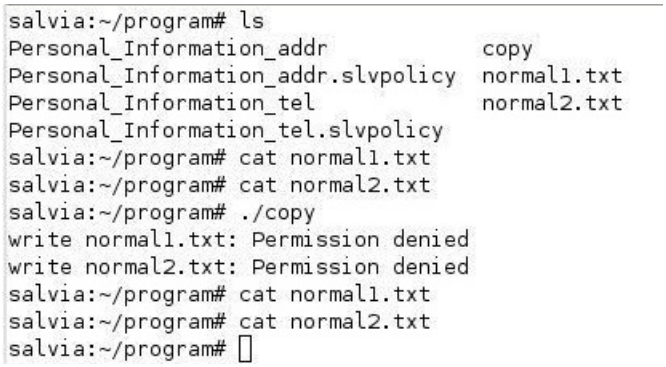

Fig. 8. Execution Result on Salvia.

Fig. 9. Execution Result on DF-Salvia.

by function *fgets* in line 18, on a process. Therefore, because the write operation of function *fputs* in both line 21 and line 23 is restricted, both normal1.txt and normal2.txt are empty. That is, while Salvia can prevent a data leakage, excessive access control occurs.

Fig. 9 shows an execution result of the test program with DF-Salvia. When function *fgets* in line 18 reads data from Personal_Information_addr, the policy is applied to the Data Flow ID 1. The operation of function *fputs* in line 21, which belongs to Data Flow ID 1, was restricted according to the policy. When function *fgets* in line 19 reads data from Personal_Information_tel, the policy is applied to the Data Flow ID 2. The operation of function *fputs* in line 23, which belongs to Data Flow ID 2, was allowed according to the policy. As a result, normal1.txt is empty and normal2.txt contains the data of Personal_Information_tel. Therefore, DF-Salvia can control access to resources based on data flows, and excessive access control, such as in Salvia, does not occur.

B. Performance Evaluation

We have measured DF-Salvia performance.

TABLE II
EXPERIMENT ENVIRONMENT.

CPU	Intel Pentimu M 1.4GHz
Memory	512MB
OS	DF-Salvia or Linux 2.6.8

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

TABLE III
PROCESSING TIME OF SYSTEM CALL.

OS	open	read	write
DF-Salvia	8.04	24.86 (9.28)	7.23 (8.57)
Linux 2.6.8	1.35	1.24	1.52

1) Experiment Description: DF-Salvia hooks open, read, and write system calls, and executes processing for access control. In this experiment, we measured the processing time on each system call in Linux and DF-Salvia for the purpose of understanding overhead on the access control. Table II shows the computer which was used in the experiment.

As the details of the experiment, we executed each system call 1000 times and calculated the average processing time. The processing time of a read or write system call on *DF-Salvia* is affected by search time required to identify Data Flow ID, which contains the library function that invokes the system call. The search time is affected by amount of data flow, which is the total number of items in the Data Flow ID list. In this experiment, we used a program which was assumed to be a practical program, whose Data flow ID list contains 5,000 items. In the measurement of read system calls, we distinguished between first reading and subsequent reading from protected file. The reason is that the processing for reading a protection policy is executed only on the first system call. Similarly, in the measurement of write system call, we distinguished the system call invoked by library functions that are Last Use Point from the system calls invoked by other function. The reason is that the processing for deleting a protection policy which is applied to data flow is executed on only the call from Last Use Point. Read/write data size affects the processing time of system calls but does not affect the overhead of access control. In this experiment, 256KB data was used.

2) Experiment Result: Table III shows the measured result of processing times on each system call. As a result of the read system call, values in brackets are the processing time in first reading, and values outside brackets are the processing time in subsequent reading. As a result of the write system call, values in brackets are the processing time on Last Use Point, and values outside brackets are the processing time for other functions.

The processing time of open system call on DF-Salvia is about 5.9 times longer than that on Linux. The difference is thought to be caused by overhead for the processing which overwrites read/write function entry in file operation table of the file object.

The processing time of read system call on DF-Salvia is about 20 times longer than that on Linux for the first read, and is about 7.5 times longer than that on Linux for subsequent read. The difference is thought to be caused by overhead for the processing of reading each protection policy. If a protection policy which defines many contents is attached to each protected file, it is thought that the difference of processing time between DF-Salvia and Linux would become large.

The processing time on write system calls on DF-Salvia is about 20 times longer than that on Linux in cases that the system call is invoked by Use Last Point, and is about 4.8 times longer than that on Linux in cases in which a system call is invoked by an other function. In addition, the processing time in read system calls increases more than that of write system calls. In read/write system calls, DF-Salvia identifies data flow, which contains the library function that invoked the system call, and determines availability of the system call execution. Additionally, in read system calls, DF-Salvia applies the protection policy to the data flow. Therefore, the processing time in read system calls increased more than that of other system calls.

As mentioned above, DF-Salvia needs processing time for execution of system calls longer than Linux. However, because the increment of overhead is limited to μs order, we consider that DF-Salvia can achieve access control which provides viable processing speed.

C. Future Issues

In line 21 and line 23 of the test program, function *setbuf* disables I/O buffer of library functions. If I/O buffer is available, write system call is invoked not by function *fgets* but by the buffer flush of function *fclose*. In this case, as the write system call is not invoked from an instruction address of Data Flow ID List (Table I), DF-Salvia cannot controlled the write system call. This increases execution overhead of programs which use I/O library functions. Solutions to this problem are shown below.

The first solution is to insert a Last Use Point which flushes the buffer into source code. As a result, because the write system call is invoked from an instruction address of Data Flow ID List, DF-Salvia can controll the write system call on the Last Use Point.

The next solution is to treat function *fclose* as a use statement. As a result, because Data Flow ID Lists have an instruction address of function *fclose*, DF-Salvia can controll write system calls which are invoked from function *fclose*.

The last solution is to analyze data flows within library functions. As a result, definition statements and use statements are not library function calls but system calls. Hence write system calls are controlled because instruction addresses of system calls are written into Data Flow ID Lists.

VI. RELATED WORK

In this section, we discuss existing work intended to prevent leakage of private information.

SAccessor [6] separates file access control mechanism from an operating system in order to protect file access control from attack which exploits security holes of an operating system. SAccessor runs two operating systems, which are work OS and authentication OS, on a machine with virtual machine monitor (VMM). The authentication OS provides a file server to the user and controls file access. The work OS provides an interface for file access. When the user seeks to access files, the work OS sends a request to the authentication OS. The authentication OS permits the requests only to add data to a system file, and controls accessing a user file based on the policy. Even if programs on the work OS are attacked, SAccessor can prevent file access by the attacker and minimize damages. However, SAccessor controls resource access only, and cannot prevent data leaks through networks and removable storage media.

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

VOFS (View-Only File System) [7] allows user to only read sensitive information regardless of user authority. VOFS runs three VM (Guest VM, SVFS VM, Domain 0 VM) on VMM. Guest VM provides an operating system environment to users. SVFS VM manages sensitive files with SVFS [8]. SVFS is a file system and achieves protection of important files. Domain 0 VM manages the whole system of VOFS. First, when the user seeks to read a file, Guest VM gets encrypted sensitive files through network and saves its file on SVFS VM. Next, if legitimacy of Guest VM behavior is confirmed, SVFS VM decodes the sensitive file, and requests Domain 0 VM to create a snapshot of Guest VM and disable device output. Finally, after the decoded sensitive file is sent from the SVFS VM to the Guest VM, the user can read data from the sensitive file. Because device output is disabled, the user can only read sensitive files. Therefore, data leaks to the exterior are not prevented. However, because VOFS does not consider creation of sensitive files and use of sensitive data, operating efficiency may be reduced and the merit of digitization may be hindered.

HiGATE [9] is PC-based High Grade Anti-Tamper Equipment which allows data to be handled without revealing the data content to administrators or users. HiGATE achieves tamper resistance by improving both hardware and software. An improved point of hardware is adoption of a function which proves that the hardware is unopened. That is, HiGATE uses tamper-resistant labels. If the tamper-resistant label is broken, efficacy of the data stored on the hardware is lost. In addition, the system introduces techniques of automatic shutdown and memory erasure when the hardware case is forced open. An improved point of software is adoption of two functions such as HDD encryption and program start-up control. HDD encryption prevents information leakage by attaching an HDD, which was detached from a computer, on to another computer. Program start-up control limits executable programs, and prevents malicious programs from stealing and fabricating data. However, because HiGATE executes predetermined programs only, the applications which the user can use are restricted. Moreover, because HiGATE developers only can register a program with HiGATE, the user cannot install or update applications easily.

VII. Conclusion

In this paper, we proposed DF-Salvia, whose access control is finer-grained than the process-based access control of Salvia. DF-Salvia distinguishes each writing of protected data to computer resources by the use of data flow information which is generated by static data flow analysis. Specifically, DF-Salvia can determines the policy for access control according to the protected data which is requested to be written to computer resources by a process. That is, DF-Salvia can prevent excess access control which may occur in Salvia.

Future work is to develop an automatic Data Flow ID List generation system and to solve problems in the access control on library functions which provide I/O buffering.

References

[1] Consumer Affairs Agency, Goverment of Japan, The Personal Information Protection Law.
(http://www.caa.go.jp/seikatsu/kojin/houritsu/index.html)

[2] Japan Network Security Association, The investigation report about an information security incident (in Japanese).
(http://www.jnsa.org/result/incident/data/2010incident_survey_PIL_v1.4.pdf)

[3] Kazuhisa Suzuki, Koichi Mouri, Eiji Okubo, Salvia: A Privacy-Aware Operating System for Prevention of Data Leakage, Advances in Information and Computer Security, IWSEC 2007, LNCS 4752, pp. 230-245, 2007.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Compilers: Principles, Techniques, and Tools (2nd Edition), Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2006.

[5] COINS project homepage, COINS Project.
(http://www.coins-project.org/)

[6] Yuji Takizawa, Kenichi Kourai, Shigeru Chiba, Yoshisato Yanagisawa, SAccessor; A Secure File Access Control System for Desktop PC (in Japanese), *Computer System Symposium*, IPSJ, pp.79-86, 2007.

[7] Kevin Border, Xin Zhao, Atul Prakash, Securing Sensitive Content in a View-Only File System, *DRM'06:Proceedings of the ACM workshop on Digital rights management*, pp. 27-36, 2006.

[8] Xin Zhao, Kevin Border, Atul Prakash, Towards Protecting Sensitive Files in a Compromised System, *SISW'05:Proceedings of the IEEE International Security in Strage Workshop*, pp. 21-28, 2005.

[9] Koji Hasebe , Ryoichi Sasaki, Development of a method for using High-Grade Anti-Tamper Equipment for privacy protection in epidemiology investigations that use data from multiple Organizations, *Proceeds of the 2nd International Conference on Information and Multimedia Technology*, pp. 99-103, 2010.

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)