Specifying and Enforcing a Fine-Grained Information Flow Policy: Model and Experiments

Valérie Viet Triem Tong[1], Andrew Clark[2] and Ludovic Mé[1]
[1]*SUPELEC*
SSIR Group (EA 4039),
Rennes, France,
{*valerie.viettriemtong, Ludovic.me*}*@supelec.fr*
[2]*Information Security Institute*
Queensland University of Technology
Brisbane, Australia
a.clark@qut.edu.au

Abstract

In this paper we present a model for defining and enforcing a fine-grained information flow policy. We describe how the policy can be enforced on a typical computer and present experiments using the proposed model. A key feature of the model is that it allows the expression of rules which detail precisely which information elements are allowed to mix together. For example, the model allows the expression of a policy which forbids a doctor from mixing the personal medical details of the patients. The enforcement mechanisms tracks and records information flows within the system so that dynamic changes to the policy can be made with respect to information elements which may have propagated to different locations in the system.

1 Introduction

We propose here a new approach for dynamic, fine-grained access control based upon modelling information flows between containers of information. In this approach, the decision about whether or not to grant access by a user u to a file f is not static but depends on the information currently contained in f. Our model separates pieces of information and containers used to hold them. In our model an information flow policy specifies the legal combinations of pieces of information and also the destination containers allowed for each legal combination of information. This approach allows the specification of a fine-grained security policy.

To enforce our information flow policy we reuse methods of dynamic data tainting to keep track of the propagation of pieces of information. More precisely, the operating system kernel tracks the flow of information to and from processes in the system through tags used to memorise the origin of information located in processes. When a process p tries to read a file f the access is granted only if the combination of the pieces of information located at this time in f is allowed to flow into p (p is viewed as a container of information). In the case where the information flow is not legal there are two possible reactions. First one can check the legality before allowing the flow and then forbidding the execution in illegal cases or one can only raise an alert when detecting an illegal information flow. In the first case the enforcement mechanism is similar to an access control reference monitor, in the second case the enforcement mechanism corresponds to an intrusion detection system parameterised by the information flow policy. Due to the technical difficulty in fully implementing a new access control paradigm in an existing operating system we choose in this work to adopt the second approach and implement our enforcement mechanism as an intrusion detection system considering, as Ko and Redmond do in [10], that an illegal information flow is an intrusion symptom. We perform here dynamic information flow tracking using tainting techniques. We designate *contents* as atomic pieces of information and we observe their propagation in the system. We also specify the possible locations for combinations of atomic pieces of information, regardless of their previous flows. This allows a fine description of the allowed information flows and we can formally prove that our information flow tracking permits the enforcement of an information flow mechanism. The number of monitored pieces of information is not fixed, so we offer the possibility of very fine-grained information flow control. Monitoring pieces of information at any level of granularity allows one to specify and enforce that certain pieces of information shall not be mixed (such as the personal data of an employee and their work-related data). We can also detect malicious activities of programs whose actions, in some cases, will violate such a policy.

In the following section we present our information flow model and describe its usage. We also address the problem of declassification of information. In Section 3 we describe how the information flow policy can be enforced at runtime, and how the *tags*, which are used to identify data associated with the information stored in a container, evolve as information propagates around the system. Sections 4 and 5 describe our prototype implementation and detail a number of experiments we have performed in a practical environment which demonstrate the utility of the approach. We also include a discussion of the practical considerations which must be addressed before adapting the proposed approach to 'real-world' applications. In Section 6 we describe the work related to ours and, finally, in Section 7 conclude the paper and discuss planned future extensions to the work.

2 Information Flow Policy

In this section we detail the model that we use to define an information flow policy expressing legal information flows. We show how the model works in practice and discuss a strategy for declassifying information.

In the proposed model of an information flow policy, we separately identify information (or contents) and containers of information existing in a computer system. Some pieces of information are explicitly distinguished as *atomic information* or *atomic contents* and are characterised by a name (also called a *tag* in this paper). The characterisation of the contents by an explicit name permits the monitoring of flows of these atomic contents starting from their original container. Containers of information are physical or logical information storage elements, for example: files, sockets or memory pages, or objects or variables in a program.

We model an information flow policy as a relation between a set of atomic contents and containers of information. We do not explicitly specify the possible paths for information, instead we list all the authorised locations and combinations for atomic contents. The policy specifies what is legal. If a combination of information, or a location for information does not appear in the policy then it is considered illegal. Formally, for a set of atomic information $\mathbb{I}$ and a set of containers $\mathbb{C}$, an information flow policy is a relation $\mathscr{P} \subseteq \mathbb{P}(\mathbb{I}) \times \mathbb{C}$, where $\mathbb{P}(\mathbb{I})$ denotes the set of all subsets of $\mathbb{I}$. A pair $(\{i_1,\dots i_j\},c) \in \mathscr{P}$ expresses that the container c is allowed to contain information arising from any subset of information $(i_1,\dots i_j)$. In other words it expresses that any combination of atomic information $\{i_1,\dots i_j\}$ can be used to forge some contents and also that these new contents can flow into c.

In this manner we explicitly specify which mixes of information are authorised and where information can be released. In Section 3 we explain how such a policy can be enforced in an information system by jointly using tags over containers of information and information flow tracking.

Traditionally security models also deal with processes that are active entities or agents ([4, 1]) responsible for all information flows. In this work we model a user u through its interface with the information system. On one hand u can generate information by himself and on the other he can access information,

	1	2	3	4
patient1	×		×	
patient2		×	×	
menu			×	
docnotes	×		×	×
		×	×	×

Figure 1: Example of specification of an information flow policy $\mathscr{P}_{ex}$.

using a screen for instance. We choose to model a user u by a pair (i_u, c_u), where i_u is atomic content and denotes any information that u can forge by himself. c_u is a container of information and denotes a virtual container of information that allows u to access information.

Processes are viewed as containers of information and we monitor the actions of processes in order to detect information flows. Each observation of a flow is taken into account by the enforcement mechanism described below.

In this model a confidentiality property of atomic content i, such as *'i cannot be accessed by the user A'*, simply means that the policy does not authorise contents arising from i to flow into the virtual container of information c_A of A. The information flow policy expresses a confidentiality property for an atomic content i towards a user A if and only if the policy has no element of the form (I, c_A) such that $i \in I$.

We consider that an integrity property is linked to a container of information: a container c of information can be modified only by authorised users. In terms of information flows this means that only information generated by authorised users may flow into the container c. When the information flow policy contains an element of the form $(\{i_A, i_B\}, c)$ it expresses that the container c can only contain information arising from i_A and i_B. If these atomic contents represent information generated by users A and B then the policy element $(\{i_A, i_B\}, c)$ expresses the integrity property *'c can be modified by users A or B'*. If the policy only contains the element $(\{i_A, i_B\}, c)$ for the container c then c can *only* be modified by users A or B.

This way of modeling an information flow policy permits the specification of all the possible locations for contents arising from atomic contents. For instance, if the private medical information of two patients (patient 1 and patient 2) are denoted by 1 and 2 and if docnotes denotes a container of information belonging to a doctor, then we can specify that the doctor is allowed to access the medical data of patient 1 and 2 and to add information generated by himself (characterised by the number 4), but he is not allowed to mix the personal data of the two patients. For that purpose we add separately $(\{1,4\},\texttt{docnotes})$ and $(\{2,4\},\texttt{docnotes})$ to the policy. In this way we explicitly forbid the combination of the two patients' medical data. The combination would have been authorised only if the policy had contained at least one element on the form $(I,\texttt{docnotes})$ such that $\{ 1, 2 \} \subseteq I$.

We consider the information flow policy described in Figure 1 using a matrix as a guided example. Containers of information (including eventually virtual containers representing a user interface) are listed in columns and atomic information (including eventually information generated by each user) in rows. A traditional file named f will be represented here through a pair (I, c) where I is a set of atomic contents and c is a container of information. From a practical point of view, each atomic content i is a unique number associated with the content of a file or a part of its content[1] at the initialisation of the system.

In this example, we expand on the one introduced previously dealing with personal data of patients

[1]for instance, for the file /etc/shadow it is interesting to differentiate each line of the file.

and access to these by a doctor. We distinguish 4 atomic contents represented by numbers 1,2,3,4. As previously 1 (resp. 2) denotes medical data of of patient 1 (resp. patient 2) . Information that can be generated by the doctor is denoted by 4. The atomic content 3 is public information of the hospital (the weekly menu, for instance). The weekly menu can be accessed by all users from the container of information menu. The menu container cannot contain any other information (the weekly menu information accessible in the file menu cannot by modified). We consider also 4 containers of information patient1, patient2, menu, and docnotes which are traditional files. patient1, and patient2 are used to store the medical data of the patient 1 and 2, respectively. These files are also allowed to contain the weekly menu. The doctor is allowed to access medical data of the patients but is not allowed to mix the medical data of the patients. The doctor can access information of the weekly menu and can use it in the file docnotes. All these constraints are expressed in the information flow policy $\mathscr{P}_{ex}$ specified in Figure 1. Each row of this matrix is an element of the policy. For a given row all the atomic contents marked with a cross are allowed to be combined and are allowed to flow into the container indicated by that row. Finally the information flow policy $\mathscr{P}_{ex}$ specified in Figure 1 is:

$$\begin{aligned}\mathscr{P}_{ex} = \; & (\{1,3\},\texttt{patient1}),(\{2,3\},\texttt{patient2}),\\ & (\{3\},\texttt{menu}),(\{1,3,4\},\texttt{docnotes}),(\{2,3,4\},\texttt{docnotes}).\end{aligned}$$

The last two elements in this policy show that the doctor can access its own information, combine it with the menu, and also separately access medical information of a patient and combine it with the menu. Any flow not listed in this matrix is forbidden. For instance the file menu cannot contain information different from the weekly menu (3).

A change in the information flow policy is directly specified by adding or deleting a cross in the matrix. The resulting changes in the information flow policy can be expressed using only addition and subtraction of pairs in the information flow policy: first we remove the pair that corresponds to the row before the modification and then add the new element. For instance, if we permit now the doctor to modify the weekly menu (for medical reasons) we will remove the element $(\{3\},\texttt{menu})$ that corresponds to the third row of the matrix and add the new element $(\{3,4\},\texttt{menu})$.

Contents existing in the system may be renamed and then considered new atomic content. This renaming operation can be used to offer a more fine-grained view of information or to declassify information. For instance, if the doctor wants now to communicate with the patient 1 – for example he wants to share with the patient his conclusion about his/her health – the doctor must relabel the information that he wishes to share. Initially, the conclusions of the doctor are information generated by himself and are denoted by 4. The doctor cannot write in the file patient1 since there is no element of the form $(I,\texttt{patient1})$ in the policy $\mathscr{P}_{ex}$, such that $4 \subset I$. Also, we cannot just add $(4,\texttt{patient1})$ to the information flow policy since every flow from the doctor to this patient would then be allowed – even flows of information that do not concern patient 1. To solve this problem the doctor creates a new atomic content with unused number i for representing his conclusions and then adds $(i,\texttt{patient1})$ to the policy. This way, only the renamed content can be disclosed and we can exactly specify where this new content is allowed to flow (into the file patient1 recording medical data of the patient). As in [14], the declassification is performed in a decentralised way: applications can rename contents and add elements to the policy. This is not a secure approach: a malicious application that wants to disclose some contents will just rename the desired content and add elements to the policy that permit the disclosure. We are aware of this shortcoming – we present here tools to perform declassification. In this work we do not study *what* information may be released or *who* releases information. Rather, we offer a precise manner to explicitly specify *how and where* information can be disclosed.

3 Enforcing an Information Flow Policy

In this section we present a general enforcement mechanism for an information flow policy expressed as proposed in the previous section. We propose a formal model for representing an information system, its containers of information and the information existing in the system. Using this formal model, we prove that if the observation of information flow is exact then the respect of the policy is ensured. In practice we monitor actions in the system that may lead to an information flow. Unfortunately, observation is inherently inaccurate. Practical aspects of implementation of the formal model are discussed in Section4.

An information system is modeled trough the 3-tuple $(\mathbb{I}_j, \mathbb{C}_j, \mathscr{P}_j)$ where $\mathbb{I}_j$ denotes the set of atomic contents that have been identified and named in the system to be monitored, $\mathbb{C}_j$ denotes the set of containers of information currently existing in the system, and $\mathscr{P}_j \subseteq \mathscr{P}(\mathbb{I}_j) \times \mathbb{C}_j$ the information flow policy effective in state $j \geq 0$. The system evolves from a state j into a state $j+1$ after the observation of an information flow that modifies the content of at least one container or after any modification of the information flow policy $\mathscr{P}_j$.

We attach two tags to each container of information $c \in \mathbb{C}_i$: an *information tag* and a *policy tag*. These tags are used to check *on-the-fly* the legality of information flows with regard to the effective information flow policy. For a container c, theses tags are denoted by $c.\mathrm{I}$ and $c.\mathrm{P}$. The *information tag*, $c.\mathrm{I}$, is a subset of $\mathbb{I}_j$ that lists atomic information used to compute the current content of c. The *policy tag*, $c.\mathrm{P}$, is a set of subsets of $\mathbb{I}_j$ used to list all combinations of atomic contents authorised to flow into c. The *information tag* evolves at each observation of an information flow and the *policy tag* evolves at each modification of the information flow policy (that affects the container).

These tags are similar to the send label and the receive label of Asbestos [14, 7]. In Asbestos, the send label of a process P represents information P has observed and the privileges associated with each of these pieces of information. The receive label of Asbestos defines what information a process can view. In our work the information tag of a container of information denotes what information is located in the container and the policy tag defines which combination of information can be located in the process. The main difference is that in Asbestos a piece of information is viewed separately whereas in our work we deal with combinations of information. In our work we can express properties such that atomic information `1` (the medical data of a patient 1) and an atomic information `2` (the medical data of a patient 2) can flow separately but not be grouped in a container of information `d` (the doctor's notes file). A second difference between these two approaches is the granularity of the description of the policy: Asbestos offers five levels of security whereas we permit dealing with any combination of atomic contents.

If the system is modeled by $(\mathbb{I}_j, \mathbb{C}_j, \mathscr{P}_j)(j \in \mathbb{N})$ then any container of information $c \in \mathbb{C}_j$ is labeled with two tags $c.\mathrm{I} \subseteq \mathbb{I}_j$ and $c.\mathrm{P} \subseteq \mathbb{P}(\mathbb{I}_j)$ which are initialised (see Section 3.2) and then updated (see Section 3.3) to guarantee the following properties:

$$c.\mathrm{I} \qquad \text{lists the origins of the content} \tag{1}$$

$$c.\mathrm{P} = \{I \in \mathbb{P}(\mathbb{I}_j) | (I, c) \in \mathscr{P}_j\} \tag{2}$$

The property (1) simply expresses that for a given container of information, $c.\mathrm{I}$ stores all the atomic information that has been used to compute the current content of c. This property will be ensured by construction at the initialisation of the tags (see3.2). When an information flow modifies the content of a container we will update the information tag to reflect the change (see3.3).

The *policy tag* stores combinations of atomic contents that can be located in each particular container. To initialise the *policy tag* of a given container c we group all pairs (I, c') in the information flow policy such that the considered container, c, is equal to c' (the second part of the pair). This way we are able to know all the combinations of atomic content that can be used to compute the current content of a container. The flow of information into a container c is legal as long as the information that arises is from atomic contents that were allowed to flow together into c. This means that the list of atomic contents used to compute the current content of c (precisely represented by c.I using property (1)) is a subset of at least one element listed in c.P.

3.1 Checking legality of an information flow

We consider a system modeled by $(\mathbb{I}_j, \mathbb{C}_j, \mathscr{P}_j)(j \in \mathbb{N})$, a container of information, c, and some information, i, arising from the combination of atomic content $i_1 \in \mathbb{I}_j, \dots, i_n \in \mathbb{I}_j$. The flow of i into c is permitted by the policy $\mathscr{P}_j$ if and only if there exists at least one element $(I, c) \in \mathscr{P}_j$ such that $\{i_1, \dots, i_n\} \subseteq I$.

We use the *information tag* and the *policy tag* to perform a dynamic verification of the legality of flows. The *policy tag* of a container lists all combinations of atomic content that can flow into the container and since the *information tag* lists the atomic contents used to compute the content, any flow of information into c is allowed if and only if c.I is a subset of at least one element of c.P, where c.I denotes the value of the information tag after the flow. This leads us to the following lemma:

Legality of an information flow We consider a system modeled by $(\mathbb{I}, \mathbb{C}, \mathscr{P})$. For any container of information $c \in \mathbb{C}$, and for any flow of information into c, where

(1) $c.\mathrm{I} \subseteq \mathbb{I}$ lists the origins of the content of c in terms of atomic content after the flow.

(2) $c.\mathrm{P} = \{I_1, \dots I_n\} = \{I \in \mathbb{P}(\mathbb{I}) | (I, c) \in \mathscr{P}\}$

the information flow is allowed by $\mathscr{P}$ if and only if the information tag of c is a subset of at least one element of the policy tag of c, *i.e.* $(c.\mathrm{I} \subseteq I_1) \vee \dots \vee (c.\mathrm{I} \subseteq I_n)$.

3.2 Initialisation of tags

At the initialisation of the system we name each atomic content that we want to monitor. The model presented here has no restriction on the number of these atomic content elements. For a very precise supervision one can associate one particular atomic content with each file at the initialisation of the system. For a personal computer this will lead to the supervision of around 10^5 atomic contents, or more. In such a situation the creation of the information flow policy may become cumbersome. In the detailed experiments we have chosen to work with only small subsets of the file system.

Starting from the information flow policy $\mathscr{P}_0$ relative to $\mathbb{I}_0$ and $\mathbb{C}_0$ we compute the tags for each container $c \in \mathbb{C}_0$ that currently contains the set of atomic information pointed out by $I_c \subseteq \mathbb{I}_0$. $c.\mathrm{I} = I_c$ is deduced during the construction of $\mathbb{I}_0$. The policy tag is computed by browsing the information flow policy and in memorising the elements of the policy dealing with c : $c.\mathrm{P} = \{I \subset \mathbb{I}_0 \,|\, (I, c) \in \mathscr{P}_0\}$. Properties 1 and 2 hold true by construction of the tags at the initialisation.

Let us consider again the example introduced in Section **??**. We consider the very small information system where we have chosen to distinguish four atomic contents: `1`, `2` , `3` and `4` and also four containers of information `patient1`, `patient2`, `menu` and `docnotes`, as previously detailed. We use the information flow policy represented by the matrix in Figure 1.

In this simple example, we consider that atomic contents `1`, `2`, `3` and `4` are the content of files `patient1`, `patient2`, `menu` and `docnotes`, respectively. The *information tags* and the *policy tags* are initialised according to the values presented in Figure 2.

As expected the *policy tag* of the container `docnotes` (belonging to the doctor) expresses that the doctor is allowed to separately access the data of patients 1 and 2 but not allowed to combine these information elements in the container `docnotes`.

File	Information Tag	Policy Tag
`patient1`	1	(1,3)
`patient2`	2	(2,3)
`menu`	3	(3)
`docnotes`	4	(1,3,4) (2,3,4)

Figure 2: Initialisation of tags in the medical example.

3.3 Evolving after observation of an information flow

An information flow from a source s to a recipient r modifies the content of the recipient and we reflect this modification in changing the information tag of the recipient to indicate the new origins of its content. When we observe that a process p reads a container of information c (as a file) we consider that there exists an information flow from c to p. The process p is also viewed as a container of information. After a flow from c to p the content of p has the same origins (in term of atomic contents) as c. On the contrary, if a process p writes a container c we consider that information flows from p to c and after the flow the content of c has the same origins as p's content.

As explained we before, we consider here the worst case: when there is an access to a container of information (in read or write mode) then there is an information flow. We detail in Section 4 how information flows are monitored in practice but we consider here that the monitor gives a complete view of the flows.

We first detail the simplest case: a unique source and a unique recipient. In this first case when the information moves from the source s to the destination we consider that the flow totally erases the previous content of the recipient. Then, after the flow, the origin of the content of r is exactly the origin of the content of s before the flow. To ensure that $s.\mathrm{I}$ lists the atomic content used to compute the current content of s (property 1) we need to update the *information tag* of the recipient container r. Since the information flow policy remains the same we do not need to update the *policy tag* of r.

When the flow appends information to the content of the recipient or more generally when we observe that information flows from multiple sources $s_1, \dots s_n$ to a recipient container r we consider that the content of r after the flow has been forged using all the source contents. Thus, we only modify the *information tag* of the recipient r – other tags remain the same – and $r.\mathrm{I} = s_1.\mathrm{I} \cup \dots \cup s_n.\mathrm{I}$. When many calculations are carried out simultaneously, we consider that the tags also propagate in parallel. For the cases when the initial contents of the recipient file r are not overwritten the information tags of the source containers (the s_i) are added to those contained in $r.\mathrm{I}$ immediately prior to the write operation.

3.4 Evolving after a modification of the information flow policy

As stated in the previous section, the information flow policy can be modified by the addition or subtraction of an element in $P(\mathbb{I}) \times \mathbb{C}$. In an access control policy a modification of permissions will affect the source container of the information: if we want to forbid a user access to particular content we change the permissions of the container where this content is supposed to be located. Let us imagine that we now want to forbid the doctor to access to medical data of patient 1. From an information flow point of view we want to prevent any flow of atomic content named `1` towards the virtual container of information `docnotes` representing the doctor's notes (in a bigger system we may also wish to prevent `1` from flowing into any container accessible by the doctor, but the approach will be the same).

If the information system was managed by a traditional access control system we will have to remove the read permission assigned to the doctor for the file `patient1` that is supposed to contain `1`. Indeed we do not know about the information really contained in the container `patient1` and also if there exists a copy of `1` somewhere else in the system, this content may be not protected. In our work we remove from the policy the association between the atomic content related to the medical data of patient 1 and all the containers of information accessible to the doctor. This way we forbid the doctor access to the forbidden data wherever the information is located. Contrary to a traditional access control system, we will change the permission of the container which is the destination of the flow. In our small example we remove the information `1` from the policy tag of the container `docnotes` and thus prevent the flow of any information arising from `1` towards `docnotes`.

More generally, if an element (I,c) is subtracted from the policy, the change only impacts the container c. After the modification to the policy, flows of any combination of elements of I into c are illicit. The set of atomic content I must be removed from the *policy tag* of the container c, such that $c.\mathrm{P} := c.\mathrm{P} \setminus \{I\}$.

If an element (I,c) is added to the policy then we have to update the *policy tag* of the container c corresponding to this new permission. We add I to the *policy tag* of the container c to maintain property 2, such that $c.\mathrm{P} := c.\mathrm{P} \cup \{I\}$.

4 Implementation and Experiments

A prototype implementation of the information flow model described above has been developed for the Linux kernel[2], this prototype is freely distributed on [8]. In addition to the instrumentation of the kernel, tools for initialising or resetting the tags and policy of files, and for displaying the tags of files and processes, have been written. These tools allow an arbitrary information flow policy to be expressed. The main tools are summarised in Table 3.

Tool	**Description**
`setipol`	Used to set the information flow policy for a container.
`lsipol`	Displays the information flow policy currently set on a container.
`setinfo`	Used to initialise the information tags for a container.
`lsinfo`	Displays the information tags of a container.
`findinfo`	Search the file system for all containers holding the specified information tags (for understanding which flows have taken place).

Figure 3: Summary of the tools developed and their capabilities.

In the following section we demonstrate how these tools are used in practice.

[2]While the implementation has no particular dependencies upon it, we used version 2.6.19.2 of the kernel running on version 5.0 of the Debian Linux distribution.

5 Experiments

We now show how the implementation that was described above can be used to implement two simple policies representing different practical situations where the approach presented in this paper will be useful. We illustrate the capabilities of the current impementation and, in each case, demonstrate how the kernel-level monitoring is able to detect the policy violations. For the first experiment we use the sample policy described earlier in the paper and summarised in Figure1. For the second experiment we show how our technique can be used to detect the propagation of a malicious shell script.

5.1 Experiment 1: Doctor's Notes

In this experiment, for simplicity, we assume that the four containers described in Figure 1 are files located in the same directory of the file system. We first need to assign an information tag to each container using the `setinfo` command. As proposed in section 3.2 this tag will refer to the initial information (contents) of the container. For example,

```
$ setinfo 1 patient1                    $ setinfo 3 menu
$ setinfo 2 patient2                    $ setinfo 4 docnotes
```

will initialise the information tags on the containers (files) `patient1`, `patient2`, `menu` and `docnotes` to `1`, `2`, `3`, and `4`, respectively. The `lsinfo` command, which by default lists the information tags for all files in the current directory (in alphabetical order) can be used to verify that the tags have been initialised correctly.

```
$ lsinfo
docnotes      4                  patient1      1
menu          3                  patient2      2
```

To initialise the policy tags the `setipol` command is used. To realise the policy described in Figure1 and Figure 2 we can use the `setipol` command as demonstrated below.

```
$ setipol -n 1 -a 1,3 patient1
$ setipol -n 1 -a 2,3 patient2
$ setipol -n 1 -a 3 menu
$ setipol -n 1 -a 1,3,4 docnotes
$ setipol -n 2 -a 2,3,4 docnotes
```

Here, the `-n` switch specifies the policy element number. Each container can have many policy elements associated with it. For example, the file `docnotes` has two separate policy elements associated with it. The `-a` switch indicates that the information tags which follow should be *added* to the specified policy element. Switches for removing information tags and entire policy elements are also available. The `lsipol` command can be used to view the policy and confirm that it has been correctly specified.

```
$ lsipol
docnotes     ( 1 3 4 )( 2 3 4 )
menu         ( 3 )
patient1     ( 1 3 )
patient2     ( 2 3 )
```

This policy allows the doctor to copy information from the menu file and either (but not both) of the two patient files into the docnotes file. It also allows information from the menu file to flow into any of the other files. It prohibits the information initially contained in the patient files from flowing to the other patient files or to the menu, and it only allows the contents of one of the patient files to be present in the docnotes file at any time.

Having initialised the files as described above, we now use a series of simple cat commands, each of which appends the contents of one file to another, to demonstrate how tags are accumulated by the destination container of information flows and to demonstrate how an alert is triggered when illegal information flows occur. In each case the syntax of the commands used is `cat file1 >> file2` which simply causes the contents of file1 to be appended to the contents of file2 (and stored in file2).

```
$ cat menu >> patient1
$ lsinfo patient1
patient1      1 3
$ cat patient1 >> docnotes
$ lsinfo docnotes
docnotes      4 1 3
$ cat patient2 >> menu
WARNING: (cat/3813) *bad info flow* in write(menu)
$ lsinfo menu
menu          3 2
$ cat patient2 >> docnotes
WARNING: (cat/3816) *bad info flow* in write(docnotes)
$ lsinfo docnotes
docnotes      4 1 3 2
```

The first cat command appends the contents of the file menu to the contents of the patient1 file (a legal flow). The subsequent lsinfo command confirms that the information tags associated with the file patient1 now includes the initial contents of the menu file. Similarly, the two subsequent commands, which respectively, append the contents of the patient1 file to the contents of the docnotes file, and list the information tags associated with the file docnotes, demonstrate another legal information flow. Two examples of illegal information flows follow. The first, which appends the contents of the patient2 file to the contents of the menu file, results in the alert shown. The alert contains the process name and process id (cat and 3813, respectively) and indicates that the illegal information flow occurred during a write to a file with the name menu. Similarly, the final two commands (and the corresponding alert) demonstrate another illegal information flow where the initial contents of the two different patient files are mixed (and stored in the docnotes file).

5.2 Experiment 2: A Malicious Shell Script

As described in the previous section, our implementation supports the specification of an arbitrary information flow policy. In practice there are a number of examples of malicious activities that an information flow policy may help to prevent or detect. One example, which we have not been able to implement in time for the submission of this paper (but which we are actively working on), is a web-based attack that is able to overwrite or modify a CGI script by exploiting a vulnerable web server. Usually CGI scripts should rarely be modified once installed, although frequently the owner of these files is same as the user whose privileges the web server runs with. A policy which forbids information flows into CGI scripts would easily detect such malicious activity.

While we have not yet completed an experiment with such a realistic example, we have been able to write a simple self-propagating, malicious script and here we demonstrate how an appropriate information flow policy could be used to detect such a script as it propagates. We stress that we are not claiming here that our approach should be considered a general technique for detecting or preventing virus-like behaviours, but rather we use this example to show that (slightly) more complicated combinations of information flows (which lead to a violation of the policy) can also be detected using the information flow model proposed in this paper.

In this example we use a simple (and relatively harmless) self-replicating shell script for the Linux system which our prototype implementation runs on. The code for the script is as follows (line numbers have been added for ease of explanation):

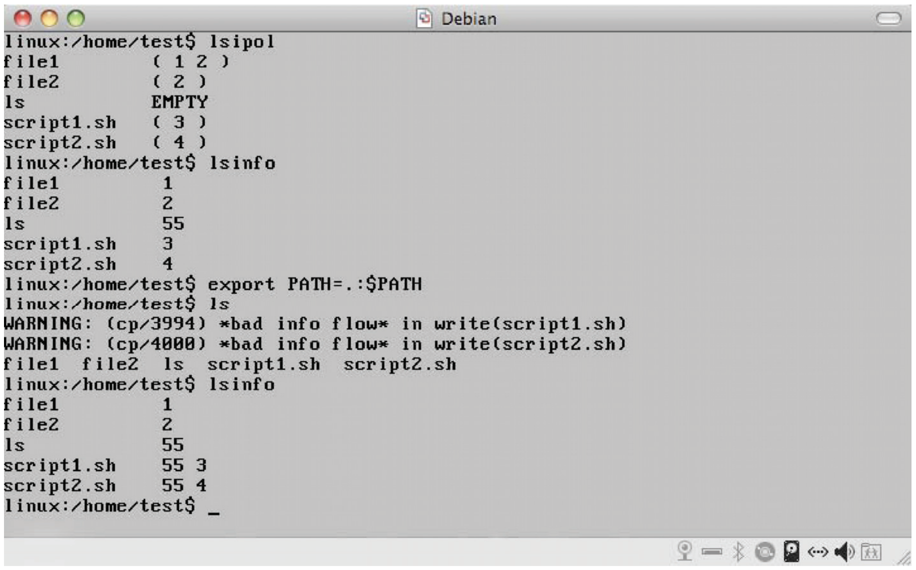
```
 1  #!/bin/sh
 2  ME=‘basename $0‘
 3  for F in * ; do
 4   if [ "$F" != "$ME" ] ; then
 5    HEAD=‘head -c9 $F 2> /dev/null‘
 6    if [ "$HEAD" = ’#!/bin/sh’ ]; then
 7     head -13 $0 > $F.tmp
 8     cat $F >> F$.tmp
 9     cp $F.tmp $F
10     rm $F.tmp
11    fi
12   fi
13  done
14  /bin/ls $*
```

The script scans each file in the current working directory (line 3) looking for other shell scripts (files which contain the string ‘`#!/bin/sh`’ at the beginning – lines 5 and 6). When such a file is found the malicious file prepends the first 13 lines of itself to the target (lines 7-10). Lines 2 and 4 perform a rudimentary check to prevent the script from modifying itself (which would cause an error). This particular script is designed to be stored in a file called `ls`. An unsuspecting user (with a ‘.’ early in their `$PATH`, for example) will invoke the script when they run `ls` from the directory containing the script (to list the files in that directory). The call to `/bin/ls` on line 14 ensures that the original call made by the user is executed and the user observes the expected behaviour.

On the prototype system, when the malicious script runs, the information tags associated with it propagate to the target file. To demonstrate this we create a test directory containing five files, initialised according to the information tags and information flow policy specified in the following table. We assign the malicious file the information tag 55 to indicate that its contents did not exist at the time the policy was specified and the contents of this file arrived on the system (and were allocated an information tag) at a later point in time. Note that in the following discussion the file `ls` is the malicious script described above.

File	Information Tag	Policy Tag
`file1`	1	(1, 2)
`file2`	2	(2)
`script1.sh`	3	(3)
`script2.sh`	4	(4)
`ls`	55	-

This simple policy allows the information initially contained in the file `file2` (denoted by the information tag `1`) to be mixed with the initial contents of, and stored in, the file `file1`. It also requires that no new information flow to either of the files `file2`, `script1.sh` or `script2.sh`. The malicious script `ls` is assumed not to have existed at the time the policy was instantiated and therefore no policy is associated with this container. Despite the fact that no policy exists on this container its information tags will be initialised by the kernel at the time the file is created.



```
linux:/home/test$ lsipol
file1         ( 1 2 )
file2         ( 2 )
ls            EMPTY
script1.sh    ( 3 )
script2.sh    ( 4 )
linux:/home/test$ lsinfo
file1          1
file2          2
ls             55
script1.sh     3
script2.sh     4
linux:/home/test$ export PATH=.:$PATH
linux:/home/test$ ls
WARNING: (cp/3994) *bad info flow* in write(script1.sh)
WARNING: (cp/4000) *bad info flow* in write(script2.sh)
file1  file2  ls  script1.sh  script2.sh
linux:/home/test$ lsinfo
file1          1
file2          2
ls             55
script1.sh     55 3
script2.sh     55 4
linux:/home/test$ _
```

Figure 4: Screen-shot demonstrating output upon execution of the malicious script.

Figure 4 shows the output of the `lsinfo` and `lsipol` commands for the test files, confirming the tags and policy and initialised as described above. The figure also shows the alerts (lines prefixed with the text ’`WARNING`’) which are produced when the malicious `ls` script is run. Finally, the figure shows the output from the `lsinfo` command which demonstrates that the information tag associated with the malicious script (`55`) has propagated to the infected script files, `script1.sh` and `script2.sh`, which now contain the tags (`3`, `55`) and (`4`, `55`), respectively.

6 Related Work

Many previous works [4, 5, 2, 1] are related to the flows of information that might occur during the lifetime of a system. Information is organised using security labels and in [3] Denning has shown that security labels representing information form a lattice structure. A lattice is a partially ordered set in which any subset has a unique least upper bound and a unique greatest lower bound. The partial order determines whether one label dominates another. Labels are attached to data containers and flows of information are only authorised between containers having equal or lower label values. R.S. Sandhu gives a synthesis of this approach in [12] using what he called a Lattice-Based Access Control Model (LBAC). Here, the models a priori identify all the possible paths for information. In our case, we identify all the possible destinations of information and not the walks. The legacy of a flow depends on the couple information/destination, and not on the path followed by the information previously.

Identifying all the possible paths would be costly. Formalizing containers of information as nodes in a directed graph and legal information flows between these containers as oriented edges in the same graph, the problem of verifying if information initially contained in one container can reach a second container is associated with the computation of a path between two nodes in the graph. In graph theory the problem can be solved by the Floyd-Warshall algorithm (FW), which computes the solution in $\mathcal{O}(n^3)$, where n is the number of nodes in the graph. In practice we cannot use FW since the number of containers of information can be very large and the edges between these nodes can be often modified. We believe that information flow control needs to offer a global view of the policy even if the management of permissions is decentralised. A static verification is not possible since it will reduce to the (FW) algorithm or a similar algorithm.

In the area of dynamic information flow control, Suh *et al.* present in [13] a tracker that detects malicious information flows at run-time. They observe that malicious actions are the result of unsanitised information that comes from untrusted input channels. Therefore, data from these input channels should be marked as *spurious* and tracked at run-time. In their approach, the security policy has to specify which input channels should be identified as spurious and what operations are authorised on spurious data. The security policy also specifies how the *spurious* tags are propagated. Contrary to ours, this approach is based on a *ad hoc* information flow policy and information is tainted only with two possible levels (spurious or not spurious).

In [15, 16] and [9] the authors propose monitoring information flows in the system in order to detect those not allowed by the information flow policy. In their work the information flow policy is deduced from access control rules. A flow of information starting from a file f_1 and ending in a file f_2 is allowed if their exists at least one user who is allowed to read f_1 and to write f_2. Starting from this specification of legality of an information flow they track information flows and raise an alert when a flow is not permitted. We also track information flows in order to detect those not allowed by the policy and one can understand our work as an intrusion detection system. However, contrary to their strategy, our information flow policy is not determined by an interpretation of the access control rules. Also in our work, a modification of the information flow policy is easily specified and enforced (which was not taken into account in [15, 16] and [9]).

In [14] Zeldovich *et al.* have presented Histar, an operating system providing strict information flow control. Histar enforces information flow using Asbestos labels [7]. Asbestos associates two labels with each process. A *send label* that represents the current contamination and a *receive label* that represents the maximum contamination that the process is allowed to accept from other processes. A label is a function which maps categories to taint levels. A category (called a handle in Asbestos [7, 6]) is an information compartment encoded using an opaque identifier delivered by the kernel. There are five taint levels namely $\star$, 0, 1, 2 and 3. The special level $\star$ represents the most privileged level, the others combine secrecy and integrity. Levels $0 \ldots 3$ represent, respectively: high integrity; default integrity and secrecy; low integrity; and high secrecy.

A file can be labeled with many categories of taint: the file label associates a taint level with each mentioned category and a default level with others. Two labels are compared by comparing each of their components. In Histar the kernel checks whether information can flow from one object to another by comparing the labels. A process P may send information to a process Q if the send label of P is less than or equal to the receive label of Q. After the flow the receive label of Q is updated to the least upper bound on the send labels of P and Q since information flows from P to Q.

In Histar applications can craft their own categories of information and the policy is not centrally specified by the administrator. As initially proposed in [11] the management of privilege is decentralised. The information flow policy is locally defined since every application can create a category and manage the permissions linked to this category: applications can freely manipulate information flow for that tag. As noted by Efstathopoulos and Kohler in [6], such decentralised privilege management diffuses the individual policies and obscures their combined effect. As a result, there is an increased risk of inconsistencies occurring in the policy as well as increased complexity in identifying and correcting such inconsistencies. In [6], Efstathopoulos and Kohler propose a policy description language that permits the expression of a policy in a human-friendly manner, and then translates the policy to the appropriate Asbestos labels. Their language permits the expression of an information flow policy as communication restrictions. First it defines compartments and then the communications rules between pairs of compartments. For each pair of compartments, communications can be forbidden, bidirectional or unidirectional, and information flows obey a transitivity property: if process *A* can send information to another process, *B*, and if process *B* can send information to process *C* then *A* can also send information to *C*. Obviously the proposed language does not offer a global view of all the possible information flows and debugging mechanisms (proposed in [6]) are needed to ensure that all possible information flows respect the security policy.

In Histar, a piece of information is viewed separately whereas in our work we deal with combinations of information. A second difference between Histar and our approach is the granularity of the description of the policy: Asbestos offers five levels of security whereas we permit dealing with any combination of atomic contents.

7 Conclusion

In this paper we have presented a model for specifying and enforcing an fine-grained information flow policy. The policy is fine-grained because it views information as atomic elements, which are distinguished from the 'containers' which hold the information (for example, files). We explain how information 'tags' associated with a container are maintained during information flows and how the policy can be enforced at the same time. The policy specifies both: (a) which atomic information elements may be combined (an *integrity* property); and (b) which users are allowed to access which information elements (a *confidentiality* property). A novel feature of the model is that it allows the policy to express restrictions on access to information elements, regardless of where they may exist in the computing environment at a point in time after the policy was specified. We also describe how a system which implements this access control system can allow changes to the policy, with respect to an information element, to apply to that information element, where-ever it may have flowed to since the time the policy was initially specified.
An implementation of the information flow model which allows a policy to be expressed on a variety of information 'containers', including files, processes, terminals (ttys) and network sockets, has been developed. We use the implementation to highlight two different cases where an information flow policy can easily express and enforce restrictions that are not well supported by traditional access control systems. While our implementation does not currently enforce the information flow policy, it records all information flows and triggers alerts in response to flows which violate the policy.
Our future work will explore in greater depth practical strategies for dealing with the *declassification* problem in the proposed scheme. We also aim to enrich the implementation further by providing more complete support for different types of information containers and by implementing the model in an *enforcement* mode which will prevent illegal information flows from occurring. The interface provided by the Linux Security Modules is likely to assist in both of these endeavours.

References

[1] D.E. Bell and L.J. LaPadula. Secure computer systems: Mathematical foundations. MTR-2547 (ESD-TR-73-278-I) Vol. 1, MITRE Corp., Bedford, 1973.

[2] K. Biba. Integrity considerations for secure computer systems. Technical Report TR-3153, Mitre, Bedford, MA, 1977.

[3] Dorothy Denning. On the derivation of lattice structured information flow policies. Technical report, Dep. of Compter. Science, Purdue University, March 1976.

[4] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[5] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[6] Petros Efstathopoulos and Eddie Kohler. Manageable fine-grained information flow. *SIGOPS Oper. Syst. Rev.*, 42(4):301–313, 2008.

[7] Petros Efstathopoulos, Maxwell Krohn, Steve Vandebogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *In Proc. 20th ACM Symp. on Operating System Principles (SOSP*, pages 17–30, 2005.

[8] SSIR group. http://www.rennes.supelec.fr/blare/, 2009.

[9] Guillaume Hiet, Valérie Viet Triem Tong, Ludovic Mé, and Benjamin Morin. Policy-based intrusion detection in web applications by monitoring java information flows. In *3nd International Conference on Risks and Security of Internet and Systems (CRiSIS 2008)*, 2008.

[10] Calvin Ko and Timothy Redmond. Noninterference and intrusion detection. In *IEEE Symposium on Security and Privacy*, 2002.

[11] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. *SIGOPS Oper. Syst. Rev.*, 31(5):129–142, 1997.

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

[13] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *SIGARCH Comput. Archit. News*, 32(5):85–96, 2004.

[14] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *OSDI*, pages 263–278, 2006.

[15] Jacob Zimmermann, Ludovic Mé, and Christophe Bidan. Experimenting with a policy-based hids based on an information flow control model. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, December 2003.

[16] Jacob Zimmermann, Ludovic Mé, and Christophe Bidan. An improved reference flow control model for policy-based intrusion detection. In *Proceedings of the 8th European Symposium on Research in Computer Security (ESORICS)*, October 2003.

Valérie Viet Triem Tong has been graduated from the Rennes University (PhD, 2003). She has been teaching in SUPELEC since 2006, where she is currently Associate Professor in the “Network and Information System Security” research group. Her research interests focus on network security.

Andrew Clark is is an Associate Professor in the Computing Science discipline at Queensland University of Technology (QUT) in Brisbane, Australia. He is also a Deputy Director of QUT’s Information Security Institute where he leads a team of researchers investigating various aspects of network security and computer forensics. His current research interests lie in the fields of intrusion and fraud detection, and network forensics. He is the author of over 70 academic publications covering various aspects of information security and is currently supervising numerous postgraduate research students in related areas. He also leads numerous industry-focused projects with large government and commercial partners in areas such as web services and control systems security, as well as fraud detection in large organisations.

Ludovic Mé is a Supélec alumnus (1987), doctor of the Université de Rennes 1 (94) and has defended a HDR in this university (2003). He is a professor at Supelec since 1998, where he is leader of a research team since 1997 (team SSIR, EA 4039). His research field mainly concerns intrusion detection, but he has interest in auto-organized networks too (ad hoc, P2P). He is author or coauthor of fifty national or international publications. He has been actively involved in a number of nationally funded projects (ANR): DICO, RAHMS, DADDi, ACES, POLUX, PLACID, LISE. He has served in many program committees of security conferences and is a member of the steering committee of RAID (Recent Advances in Intrusion Detection, program comity chair in 2001 and general chair in 2009) and of the conference CESAR from the DGA. He is also a member of the editorial board of the ”European Research Journal in Computer Virology” (Springer). He serves a few advisory boards: the board for the security activities of the DGA, the board for the security activities EDF, and the French defense scientific board. He has been a member of the selection committees for several ANR programs: Telecom program (2005 et 2006), SetIN program (2006), and SeSur program (2007). Ludovic Mé has spent in 2001 seven months in the ”Security Lab” of the University of California at Davis. He teaches intrusion detection and security in several institutions: Rennes University, Limoges University, Télécom Bretagne, ENSAI, and, of course, Supélec. Here, he is in charge of the last year option dedicated to information systems security.