INFORMATION FLOW ANALYSIS FOR FILE SYSTEMS AND DATABASES USING LABELS

Ehud Gudes, Luigi V. Mancini, and Francesco Parisi-Presicce

Abstract The control of information flow has been used to address problems concerning the privacy and the secrecy of data. A model based on decentralized labels extends traditional multilevel security models by allowing users to declassify information in a distributed way. We extend this decentralized labels model developed by other authors by addressing specific issues that arise in accessing files and databases and in general in I/O operations. While retaining the support for static analysis, we also include run-time checks to allow declassification with "controlled information leakage".

1. INTRODUCTION

One of the ways used to address the problem of privacy and secrecy of data has been through the use of models for a precise control of the propagation of information [1–3]. Several information flow models have been developed over the years, mostly for traditional multilevel systems [11]. Some of the more recent ones are geared toward object-oriented database systems [9, 10], and toward a decentralized control of access information for a distributed environment [6]. Our work is based on ideas of [5] and [6] and focuses on alternatives to that notion of declassification, especially as it relates to flies and databases.

The decentralized label model of [5] is more complex than traditional models used in classical multilevel security policy. A *decentralized label* reflects the flow policies of the individual principals on the data so labeled, and the use of the labels is such that the different policies are satisfied simultaneously. Confidentiality is then protected even in the context where the principals distrust each other or their respective code. Note that policy here refers to a single data entity and not to the behavior of the system as a whole. This decentralized label model is based on annotated program code that is statically analyzed. It provides a set of rules that a program must follow to prevent leakage of information. A compiler statically checks the flows of information in a program and allows its execution if all the *re-labeling* in the program are safe, i.e., legal according to the rules. In traditional information flow control models, declassification of information is outside the model and is performed by a trusted subject. This aspect makes such models unsuitable in a decentralized environment. In the decentralized label model of [5, 6], limited forms of declassification are allowed to be included explicitly by the programmer, provided that other principals' policies are not affected.

Our objective is to relax some of the restrictions and consider I/O channels, files and databases at run-time, and restricted forms of controlled information leakage.

If the program is compiled when the files to be used at run-time are known, no later check is needed. If the files or the file labels change, it is necessary to recompile the program to verify that the rules for declassification are followed. An alternative to this approach is to use the compilation to produce constraints on the labels of the files to be used: at run-time, there is a check of permissions to verify that the labels of the files actually used satisfy the compile-time constraints. If the constraint is, for example, the requirement that the label of the output file be no less restrictive than the label of the output variables of the program, the problem reduces to the safe relabeling in [6]. But the constraints can be of a more general nature (see Section 3) reflecting a more relaxed policy for information leakage. All this is done independently of the subject executing the program. The security level of the subject executing a program should be taken into account, especially when dealing with I/O channels. This is discussed in this paper.

Declassification can occur at two levels. By the use of the appropriate instruction, declassification takes place inside the program and deals with the labels of variables [5]. The external modification of the label of a file also may constitute a declassification of the information in the file, if for example one of the owners of the file adds a reader to his own policy. This kind of declassification takes place in the administration of files.

Myers and Liskov incorporated the approach and the ideas of [6, 5] into the specific programming language framework of Java, and called their extended Java language *Jflow* [4]. This extended language allows both compile-time and run-time label checking, supports the definition of label data types, and provides for automatic label inference.

Samarati et al. presented a related model in the context of *Object-oriented databases* [9]. The model provides only for run-time checking by accumulating the flow during program execution. It checks all the flows between objects and between calling and called methods. The checks are done by a special trusted system component called the *Message filter.* The model was later extended in [10] to support exceptions. In a recent paper [12] Chen, Wijesekera, and Jajodia show how different flow models like Denning's lattice model, and Myers distributed label model can be formalized using a single Prolog-like language *Flexflow.* A different approach to flow control in a web environment was suggested in [13]. In this approach, the flow is controlled by a special encryption based web viewer, but the flow policies supported are quite limited.

In the rest of this paper, Section 2 overviews the basic notions of the model based on distributed control via labels and policies in [5, 6] and proposes a relaxation of this model. Section 3 extends the use of the labels to files and I/O operations, and is also concerned with the declassification process, before and after compilation. Section 4 deals with database access and update with views and labels. Section 5 contains a summary and points to future work.

2. THE LABEL MODEL

In this section, we briefly describe a label model very similar to the one proposed in [5]. The presentation is slightly different.

The model is based on a notion of *label* that is used to categorize different entities and to control the transfer of information between two such entities. A label can be viewed as a type and rules are described to define *compatible* types. We first present a simple version with no relation among the principals, and then we extend it to consider a hierarchy based on the notion of *acts for.*

Labels are associated with values (computed or read by a program), variables (declared and modified in a program), I/O channels (used by programs), files and views of a database. The compatibility between types determines, for example, whether a variable can hold a value or can be used to read a file, or whether a value can be written into a file or into a database.

Given a set of *owners* and a set of *readers,* not necessarily distinct, a *label* is just a set of pairs (o, r) with o an owner and r a reader. A pair (o, r) in the label L indicates the willingness on the part of the owner o to grant reading rights to r for any information labeled L. If there are several owners in the label, the access to information labeled L depends on the combination of the different authorizations.

For a label L

$$owners(L) = \{o : \exists\, reader\ r\ such\ that\ (o, r) \in L\}$$
$$readers(L, o) = \{r : (o, r) \in L\}$$

Moreover, the following set definitions will be useful in the rest of this paper:

$$INT(L) = \bigcap\{readers(L, o) : o \in owners(L)\}$$
$$UN(L) = \bigcup\{readers(L, o) : o \in owners(L)\}$$

Note that *INT(L)* is what in [6] (without the principal hierarchy) is called *effective-readers(L).*

A label is completely characterized by *owners*(L) and *readers* (L, o) for each $o \in owners(L)$.

If $owners(L) = \{o_1, o_2, o_3\}$, $readers(L, o_1) = readers(L, o_2) = \{r_1, r_2\}$ and $readers(L, o_3) = \{r_1, r_3\}$, then it is convenient to denote L by $\{o_1$:

$r_1, r_2;\ o_2 : r_1, r_2;\ o_3 : r_1, r_3\}$ as in [5, 6] rather than by $\{(o_1, r_1), (o_1, r_2), (o_2, r_1), (o_2, r_2), (o_3, r_1), (o_3, r_3)\}$.

Based on the above conservative definition of access, there are two ways to declassify information through a modification of the label: either by reducing the set *owners*(L), that is by removing an owner (more precisely, by removing all the pairs with the specific owner as first component) or by adding rights by augmenting one of the sets *readers*(L, o)(more precisely, by adding new pairs (o, r) for an existing o). For example, a data x labeled $\{o_1 : r_1, r_2; o_2 : r_1, r_2; o_3 : r_1, r_3\}$ can be declassified to allow r_2 to access it either by removing $(o_3, r_1), (o_3, r_3)$ (i.e. the entire o_3 entry), or by adding (o_3, r_2). In both cases r_2 becomes part of the intersection of the readers set of L and is authorized to access x according to the most conservative policy mentioned above.

Labels can be 'ordered' by defining $L_1 \sqsubseteq L_2$ if L_2 is at least as restrictive as L_1 in granting permissions. More formally:

DEFINITION 1 (SIMPLE ORDERING) $L_1 \sqsubseteq_s L_2$ *if and only if* $owners(L_1) \subseteq owners(L_2)$ *and* $\forall o \in owners(L_1)$, $readers(L_2, o) \subseteq readers(L_1, o)$.

A simple fact that follows from the above definition is:

PROPOSITION 2 (LABELS INTERSECTION) *If* $L_1 \sqsubseteq_s L_2$ *then* $INT(L_2) \subseteq INT(L_1)$

2.1 Ordered owners and readers

To allow indirect access rights based on inheritance, owners and readers are organized in a *principal hierarchy,* i.e., structured using a reflexive and transitive relation ρ.

A pair $(o_1, o_2) \in \rho$ indicates that the owner o_1 can prevent the owner o_2 from assigning access rights; a pair $(r_2, r_1) \in \rho$ indicates that r_1 can read whatever r_2 can read. This hierarchy determines, for each label L, a set $E(L)$ of *effective flows* (o, r) obtained by adding to L all pairs (o, r) implicitly defined (via ρ) by L. To this extent, we define a function ADD_ρ on sets S of pairs (o, r) by letting:

$ADD_\rho(S) = \{(o,r) : \exists\, o' \text{ with } (o',o) \in \rho,\ (o',r) \in S\} \cup \{(o,r) : \exists\, r' \text{ with } (r',r) \in \rho, (o,r') \in S\}$

Since ρ is reflexive, $S \subseteq ADD_\rho(S)$. As a function of S, $ADD_\rho(S)$ is monotonic with respect to the Simple Label Ordering, i.e., if $S \sqsubseteq_s S'$ then $ADD_\rho(S) \sqsubseteq_s ADD_\rho(S')$. The function $ADD_\rho(S)$ is also monotonic in ρ (i.e., if $\rho \subseteq \sigma$, then $ADD_\rho(S) \subseteq ADD_\sigma(S)$), but this fact will not be used here.

Let us denote S by $ADD^0_\rho(S)$ and $ADD^{i+1}_\rho(S) = ADD_\rho(ADD^i_\rho(S))$. Since the sets of owners and readers are finite, the set $\{ADD^i_\rho(S) : i \geq 0\}$ is finite, with $ADD^k_\rho(S)$ its largest element for some k. The *closure* of S with respect to ρ, denoted by $Cl_\rho(S)$, is by definition this $ADD^k_\rho(S)$. We denote the set of readers defined by the above closure as the *effective set of readers.*

Definition 1 considers all owners and readers unrelated in the hierarchy. By taking into account the principal hierarchy ρ for owners and readers, the definition extends to:

DEFINITION 3 (HIERARCHICAL LABELS ORDERING) $L_1 \sqsubseteq L_2$ *if and only if* $\forall o_1 \in owners(L_1)$, $\exists o_2 \in owners(L_2)$ *such that* $(o_2, o_1) \in \rho$ *and* $\forall r_2 \in readers(L_2, o_2)$, $\exists r_1 \in readers(L_1, o_1)$ *such that* $(r_2, r_1) \in \rho$.

It is straightforward to verify that $\sqsubseteq$ is reflexive and transitive. If $L_1 \sqsubseteq L_2$ and $L_2 \sqsubseteq L_1$, then $Cl_\rho(L_1) = Cl_\rho(L_2)$ but it is not necessarily the case that $L_1 = L_2$.

EXAMPLE 4 *The set of owners consists only of* o_1 *and* o_2, *unrelated by* ρ, *and the set of readers is* $\{r_1, r_2, r_3, r_4\}$ *with* $(r_3, r_1) \in \rho$ *and* $(r_4, r_2) \in \rho$. *Define* $L_1 = \{o_1 : r_1; o_2 : r_2, r_4\}$ *and* $L_2 = \{o_1 : r_1, r_3; o_2 : r_2\}$. *Then, by definition,* $L_2 \sqsubseteq L_1$ *since* $owners(L_1) = owners(L_2)$, ρ *is reflexive and* $(r_4, r_2) \in \rho$. *Similarly* $L_1 \sqsubseteq L_2$ *using* $(r_3, r_1) \in \rho$. *Furthermore,* $Cl_\rho(L_1) = Cl_\rho(L_2) = \{o_1 : r_1, r_3; o_2 : r_2, r_4\}$. *Obviously,* $L_1 \neq L_2$.

Note that taking the hierarchy into account, $Cl_\rho(L)$ is a simpler way of defining what is called $\mathbf{X}L(L, \rho)$ in [6].

PROPOSITION 5 (CLOSURES) *If* $L_1 \sqsubseteq L_2$ *then* $Cl_\rho(L_1) \sqsubseteq Cl_\rho(L_2)$

The intuition behind $L_1 \sqsubseteq L_2$ is that entities labeled L_1 can be safely re-labeled L_2, safely in the sense that under the new label L_2, the number of readers with access to the information is not increased.

So the process of *declassifying* information consists of re-labeling the information from a label L_2 to a label $L_1 \sqsubseteq L_2$. Note that the declassification could only be apparent and not real, since we may have $Cl_\rho(L_1) = Cl_\rho(L_2)$ even if $L_1 \sqsubseteq L_2$ and $L_1 \neq L_2$.

We argue, and will discuss it in the next section, that in dealing with files and I/O operations, $L_1 \sqsubseteq L_2$ may be too restrictive as a notion of safe flow and could be relaxed. The confidentiality of the information labeled with L_1 is preserved by re-labeling it with L_2 also if $UN(L_2) \subseteq INT(Cl_\rho(L_1))$, that is if any potential effective reader in any policy in L_2 is considered trusted (i.e., contained in the set of effective readers) in every policy of L_1.

DEFINITION 6 (SAFE RELABELING) *An item labeled L_1 can safely be relabeled L_2 if either*

- $L_1 \sqsubseteq L_2$ *or*
- $UN(L_2) \subseteq INT(Cl_\rho(L_1))$

The constraint $UN(L_2) \subseteq INT(Cl_\rho(L_1))$ is neither more restrictive nor more relaxing than $L_1 \sqsubseteq L_2$ as the next two examples show. So the relaxation consists of requiring either one of the two conditions to allow the relabeling.

EXAMPLE 7 *The set of owners consists of o_1, o_2 and o_3, unrelated by ρ, and the set of readers is $\{r_1, r_2, r_3\}$.*
Define $L_1 = \{o_1 : r_1, r_2; o_2 : r_1, r_3\}$ and $L_2 = \{o_3 : r_1\}$. Then, by definition, $L_1 \sqsubseteq L_2$ does not hold since there is no "acts for" relation between o_3 and either o_1 or o_2. But the only reader authorized by o_3 is also authorized by both o_1 and o_2 and so $UN(L_2) \subseteq INT(Cl_\rho(L_1))$ holds.

The next one shows that the new condition would be too restrictive if used alone.

EXAMPLE 8 *The set of owners consists of o_1, o_2, o_3 and o_4, with $(o_3, o_1) \in \rho$ and $(o_3, o_2) \in \rho$, and the set of unrelated readers is $\{r_1, r_2\}$. Define $L_1 = \{o_1 : r_1; o_2 : r_1\}$ and $L_2 = \{o_3 : r_1; o_4 : r_2\}$. Then, by definition, $L_1 \sqsubseteq L_2$ since o_3 acts for both o_1 and o_2, but $UN(L_2) \subseteq INT(Cl_\rho(L_1))$ does not hold because of the policy of o_4.*

In the rest of this paper, we will write $Cl(L)$ instead of $Cl_\rho(L)$ when the hierarchy does not play a role or is implied by the context.

2.2 Related Issues

In this section, we discuss some of the issues relevant in the model to access files, databases and in general to perform I/O operations.

Consistency A first issue is the uniform and consistent treatment of files, I/O operations and/or channels. For example, in the model(s) presented by Myers and Liskov [6] the writing to an output channel is allowed only if the set of readers of the channel is a subset of the *effective* set of readers of the value written to the channel. In [5] the writing to a channel is allowed only if the label of the channel is more restrictive than the label of the value written to the channel. Now these definitions are not equivalent. For example, if a value label is $l(v) = \{o_1 : r_1, r_2; o_2 : r_1\}$ and the channel label is $l(f) = \{o_1 : r_1\}$ then, according to the first definition, writing the value v on the channel f is allowed, since r_1 is in the effective set of readers, while according to the second definition, writing is not allowed since the channel label is less restrictive (an owner was removed). Our approach is closer to the first definition.

Declassification In [5] there are two concepts related to Declassification: the *Authority* predicate, and the *Declassify* operator. The Authority predicate specifies which principal can execute an application process and allows this correlation to be checked at *run-time.* Once the application process is running, then the explicit Declassify operator might be invoked with the desired label as a parameter. Since in [5], the Declassify operator can only remove policies which are owned by the principal listed in the Authority predicate, this check can be done at *compile-time.* No further check is done at run-time! This, however, presents a subtle problem with the files access and the I/O channels, as exemplified by the following taxpayer example from [5]. In this example, a tax-payer Bob likes to use a general tax program that accesses a proprietary database to which only the Preparer has access. The compiler does not allow writing back into Bob's file because the label was increased to include the Preparer. So the only way to allow this example to work, is to allow a declassification at run-time. In [5] the validity of the Declassify operation is checked at compile time, and since both the Authority predicate and the Declassify operator contain the explicit label "Preparer", the check succeeds.

This, however, raises two problems. First, this type of check limits the generality of the Declassify operator, since it requires that all parameters to Declassify and Authority be explicit at compile-time. Second, it also raises the problem of *Covert channels,* since anybody who can bypass the Authority predicate, can read the information from the Preparer database, can declassify it and can leak it via a covert channel. A more restrictive approach is desirable either at compile-time or at run-time.

The above issues call for a more precise definition of I/O operations and of the operation of declassification in the decentralized labels model, and represent the first goal of the present paper. The second goal is to extend the model to the database case.

3. FILES AND I/O CHANNELS

3.1 Compile-time checks

In the previous section, we mentioned two possible interpretations of permissions to write to a file and I/O operations. This section analyzes the issues and proposes our solution based on the *intersection* interpretation for writing. This solution for files and I/O channels are also extended to deal with administration, delegation, database operations and views in the following sections.

We assume that files have labels, so that a file f has the label $l(f)$, and that v is a variable with label $l(v)$. that is,.

We define the permission to write to a file in terms of the two sets *INT*(*L*) and *UN*(*L*), defined in Section 2.

DEFINITION 9 (READ FROM A FILE) *A read operation* `read(f,v)` *of a file f succeeds if $l(f) \sqsubseteq l(v)$,*
that is, `read(f,v)` *is allowed if the label $l(v)$ of the variable is at least as restrictive as the label $l(f)$ of the file.*

DEFINITION 10 (WRITE TO A FILE) *A write operation,* `write(f,v)`, *to a file f is allowed if $UN(l(f)) \subseteq INT(Cl(l(v)))$*
that is `write(f,v)` *succeeds if all the readers specified in the file label $l(f)$ are contained in the intersection of all the effective readers in the variable label $l(v)$.*

EXAMPLE 11 (APPLICATION OF FILE RULES) *If $l(v) = \{o_1 : r_1, r_2;\ o_2 : r_1\}$ and $l(f) = \{o_2 : r_1\}$, then* `write(f,v)` *is granted, even though $l(f)$ is less restrictive than $l(v)$ since an owner is removed. Also, when $l(f) = \{o_3 : r_1\}$, the operation* `write (f,v)` *would be allowed.*

Note that having on the file f the label $\{o_3 : r_1\}$ is useful in practice. Consider the two operations in sequence `write(f,v)`, `read(f,u)`, *where label of u is $\{o_3 : r_1\}$. These operations appear equivalent to the assignment* `u = v`. *However, the subsequent read from file f into the variable u with label $\{o_3 : r_1\}$ is granted, even if o_3 is not an owner of v, while the assignment* `u = v` *is denied since $l(v) \sqsubseteq l(u)$ does not hold.*

If we allow the `write(f,v)` *only to a file with at least as restrictive a label, the subsequent read in u could not be performed, even if r_1 belonged to $INT(Cl(l(v)))$.*

So far, all checks done at compile-time are similar to static type-checking: they make sure that the information flow along the valid paths is correct. In particular, the compilation of program *Prog* produces a set of constraints on the labels of the files and I/O channels that *Prog* can access. A run-time mechanism is needed to enforce that only the read and write operations on files and I/O channels that satisfy these constraints are allowed. Note that the run-time checks may be independent from the credential of the subject executing *Prog*. To clarify these concepts in the following we take a closer look at run-time checks.

3.2 Run-time checks

If all checks were made only at compile time and *any* subject could run the application, then the whole label structure could not guarantee the correct information flow. For example, if a subject logs into a system to run an application to read and write confidential data on a video-terminal, a run-time check is needed to compare the label of the I/O channels associated with the subject with the label of the input/output variables declared in the application. Therefore critical labels must be kept at run-time and some correlation must be checked between the subject running the application and the subjects appearing in the label of the variables.

A read operation `read(IN,v)` from an input channel *IN* succeeds if $l(IN) \sqsubseteq l(v)$, that is, `read(IN,v)` is allowed if the label $l(v)$ of the variable is at least as restrictive as the label $l(IN)$ of the Input channel. Note that the label $l(IN)$ may be determined by the authorization subsystem at run-time on the basis of the credentials of the subject associated with the channel *IN* (a typical example of such a channel is the subject's keyboard). Of course, also the label $l(v)$ of the variable where the data is read from *IN* must be kept at run-time.

A write operation `write(OUT.v)` to the output channel *OUT* is allowed if $UN(l(OUT)) \subseteq INT(Cl(l(v)))$

Note that the label $l(OUT)$ may be determined at run-time according to the credentials of the subject which is associated with the channel *OUT* (a typical example of such a channel is a printer or a video-terminal).

Consider the sequence of the two operations `read(f,v); write(OUT,v)` executed by a subject to display the content of a file f on a video-terminal associated with the output channel *OUT.*

For the operation `read(f,v)` to be granted, we must have $l(f) \sqsubseteq l(v)$, which, by Propositions 2 and 5, implies $INT(Cl(l(v))) \subseteq INT(Cl(l(f)))$. If also the `write(OUT,v)` is granted, then it must be $UN(l(OUT)) \subseteq INT(Cl(l(v)))$. Hence $UN(l(OUT)) \subseteq INT(Cl(l(f)))$ holds. This means that the union of all the readers in the label of the Output channel is contained in the intersection of the readers of file f; that is a subject that displays a file f must be authorized by all the owners of f.

THEOREM 12 (SAFE INFORMATION FLOW) *Under the conditions in Def. 6, information flow from a file (channel) to another file (channel) is safe.*

The proof is straightforward and will not be detailed here due to space limitations.

3.3 The problem of Covert Channels

In the discussion on I/O channels above, we assumed that the program contains read or write operations to I/O channels, and the subject *running* the program is associated with the I/O channels accessed by the program and this correlation is checked at run-time.

A different situation occurs when the program performs only I/O operations on files. Who can execute such a program? What run-time checks are needed? In this situation, an illegal information flow outside the system may occur due to covert channels.

In the following discussion, we assume that, from the point of view of the operating system, the subjects mentioned below (i.e. r_1, r_2, r_3, r_4) have the right to execute the program.

Consider a program *progR* that executes the operation `read(f,v)`, that is, *progR* attempts to read a file f, with label $l(f) = \{o_1 : r_1, r_2; o_2 : r_1, r_3\}$ for example, into a local variable v. Subject r_1 should be allowed to execute the program and read f, since r_1 is authorized to read the file by all the owners.

Can a subject r_4 that does not appear in the label $l(f)$ be allowed to execute *progR* ? Note that if we allow r_4 to execute `read(f,v)` and read from file f to v, r_4 cannot write v into r_4's private files or other I/O channels because of the compile-time checks. However, r_4 could try to leak the contents of v through a covert channel. For example, r_4 could design and execute a program *ProgCC* that first reads a sensitive file f and then leaks its contents via a covert channel implemented by *ProcCC* itself. But, subject r_3 should be allowed to execute *progR,* since r_3 is in the policy of o_2. With such a restriction, the information that r_3 might leak is the one that r_3 is authorized to read, and could be leaked by r_3 anyway employing any other malicious covert channel; while without our restriction r_4 could leak the content of a file that r_4 is not authorized even to read. The above restriction must be enforced at run-time. It is worth noting the inevitability of covert channels whenever run-time checks are present. This problem arises because the run-time check may fail, and the fact of failure (or its absence) may lead to a leak of information via covert channel. Thus, limiting the class of users that can execute the program, reduces this risk of covert channels.

Consider another program *progW* that executes the operation `write(f,v)`, that is *progW* attempts to write into a file f. Should we check which subject can run *progW* as well? This is not necessary, since the compile-time checks enforce the outwards information flow defined by the labels. Note, however, that one may employ additional *access control* policies, to limit the set of subjects who can write into a file. See also our discussion of Database access in Section 4.

To summarize this point, we check at run-time the subjects running an application for two purposes: 1) if the subject is not an owner, then limiting the execution only to subjects appearing in one of the label entries of the input channels has the advantage of restricting leakage via *covert channels;* 2) if the subject running the application is an owner, then some form of declassification is possible while keeping control over the dissemination of the information (see next section).

3.4 Declassification

Our approach to declassification is that only *owners* of I/O channels and files can declassify information. This choice is inspired also by the *originator controlled release* model used by the DoD/Intelligence community.

The check on information declassification can be enforced both at compile-time or at run-time, based on the *Authority* predicate and the *Declassify* operator, discussed in Section 2. In addition, we want a principal specified in the authority predicate to be allowed to perform declassification only of information he owns. If the *Authority* predicate is employed to specify, in the code, the list of principals that can execute a program, then we can check at compile-time that every principal in that list is contained in the intersection of *all* the owners of all the files and input channels read by the program. If this is the case, then the *Declassify* operators present in the program are allowed, otherwise a compile-time error is returned. Of course, the compile-time check of declassification is quite restrictive, since not every execution will involve reading from all files. The difference between the above check and the check in [5] is that now the parameter to Declassify need not be explicit, thus the program may be more general (even though it involves more overhead...).

PROPOSITION 13 *The "only owners" compile-time check is at least as restrictive as the check in [5].*

The proof is direct. Every explicit principal in Declassify of [5] must have an owner policy, which means it is the owner of one of the read files, which is a necessary condition to being in the intersection of all owners of read files.

A less restrictive (and more precise) approach involves a run-time check. However, a run-time check requires to *collect* the information flow at run-time (similar to what is done by the *Message filter* approach of Jajodia et al [9]), and to check each invocation of a declassify operator(*DO*) against the collected flow up to the invocation point of *DO* (making sure that the principal executing the program is actually in the intersection of the set of owners of the current flow). Note that compared to the overhead in [9], there is less runtime overhead here, since we only need to store labels of files and input channels.

Summarizing, a compile-time approach restricts to the worst case the set of principals who can run a declassification program, while a run-time approach has a higher overhead, since it implies collecting the information flow and checking it for each declassify operator invoked.

The next example shows the interaction of the file operations and of the declassification issue in this model.

EXAMPLE 14 (UPDATING A CUSTOMERS-ACCOUNT FILE) *Consider a customer accounts file in a database and a customer who wants to read his account data, and update some personal information. The customer subject is*

o_2 and the customer's private file is owned only by o_2 and only o_2 can read the file, that is, the private file is labeled by $\{o_2 : r_2\}$. Similarly, the customers accounts file in the database is owned by the database administrator o_1, and its label is $\{o_1 : r_1\}$, so only o_1 can read/write to that file. Here are two scenarios (see Figure 1):

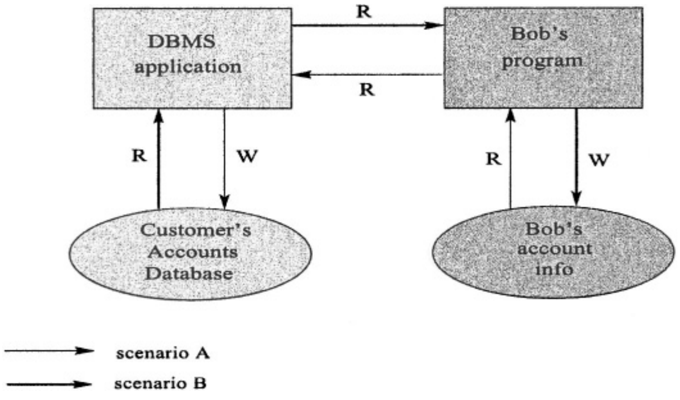

Figure 1. Accounts database example.

Scenario A

Owner o_2 wants to update his account in the database with information from his private file. Then, o_2 reads his private file into a variable p with label $\{o_2 : r_2\}$, and variable p is passed as a parameter to a program QW that updates the account data of o_2 into the customer account file. Program QW manages an array of slots V[n], each slot is labeled statically with label $\{o_1 : r_1; o_i : r_i\}$, where o_i, r_i represents the i-th customer. QW compares the parameter label of p with the label of the field V[j], using a switch or an if statement (like in the JFLOW language [4]), and sets $V[j] = p$ if the two labels agree. Now, assume that V[j] has been updated on behalf of o_2, and has been labeled $\{o_1 : r_1; o_2 : r_2\}$, to allow the actual update of the database. Then, o_2 should declassify V[j] and set it to $\{o_1 : r_1\}$ only.

Next, the program QW invokes another program Q' with the privileges of o_1 (in the Unix system the program Q' could be implemented as a **setuid-to-o_1** *process) and passes to Q' the parameter V[j] to update the database. Since the label of V[j] now equals that of the customer accounts file owned by o_1, the update of the database can be performed.*

Scenario B

Owner o_2 wants to read his account information from the database into a private file. Subject o_2 may call a program QR with **setuid-to-o_1**, *passing, as a parameter, a variable p labeled $\{o_2 : r_2\}$. QR can read the customer accounts database with o_1 privileges, and can enter the information into a QR local* *variable x with label $\{o_1 : r_1; o_2 : r_2\}$. Then, QR declassifies this information removing o_1 from the label of x, which becomes $\{o_2 : r_2\}$. Program QR can do this since QR is running on behalf of the owner o_1 of the database. Next, the program QR sets p with the information read from x and returns p to o_2. Now o_2 can write the information back into his private file, since the label of p is $\{o_2 : r_2\}$.*

The above two scenarios show that even under the restriction that only owners can declassify information, reads and updates of personal information into a shared database can be performed.

3.5 Administration and Delegation

In this section we discuss the policies to manage and administer file labels. Suppose a file has a label $\{o_1 : r_1\}$. The following are possible administrative operations that can change labels of files:

1 Grant read - any listed owner can perform this operation, e.g., owner o_1 can change the label to $\{o_1 : r_1, r_2\}$, granting read privileges to r_2.

2 Grant ownership - this operation adds an owner to the label. Any listed owner can perform this operation, e.g., owner o_1 can change the label to $\{o_1 : r_1, r_2; o_2 : r_2\}$. Of course a more restrictive policy is also applicable, where only a specific owner (e.g. the super-user) can perform a grant ownership.

3 Revoke read access - any listed owner can perform this operation by deleting a reader from his list.

4 Revoke the ownership of another owner - in our model all owners are equal, so no owner can revoke another one, unlike the SQL database, where the Grant hierarchy is preserved.

5 Revoke self-ownership - we call this operation *Administrative Declassification* that any listed owner can perform by deleting himself from the owner field of the label.

3.5.1 Uses of administrative declassification. Administrative declassification is necessary to manage file labels. One interesting use of it is to deal with the problem of *NON-originator controlled release.* In an *originator controlled release* policy, the originator always maintains control on the information flow, even if it is copied to other files. This is the case also with the basic labels model, where the owners always maintain control. Now, sometimes an owner wants to have another user copy his file and to remove himself as responsible for this file.

Suppose o_1 is the owner of $f1$ with label $\{o_1 : r_1\}$ and wants to give to o_2 a file $f2$, a copy of $f1$, without o_1 appearing as an owner of $f2$. How can this be done in our model? There are two ways:

- Subject o_1 creates a file $f2$ with both o_1 and o_2 as owners and adds r_2 as a reader in the label of $f1$. Now subject o_2 can copy the file $f1$ to $f2$, since the label of $f2$ is more restrictive than the label of $f1$. Finally, o_1 removes himself as an owner of $f2$ and removes o_2 as a reader from $f1$.
- Subject o_2 creates a file $f2$ with only o_2 as an owner. Subject o_1 adds o_2 as an owner for $f1$ temporarily, and then o_1 reads file $f1$ in a local variable, declassifies the data by removing his owner entry (i.e. the data are temporarily owned only by o_2) and writes it into $f2$. Finally, o_2 uses administrative declassification to remove himself as an owner of file $f1$.

Basically, the difference between the two scenarios is on who does the copying, but either way, o_1 must be involved into resigning from the originator control on his data. That is, o_1 cannot remove his own responsibility without performing an explicit operation.

4. DATABASES AND VIEWS

The decentralized labels approach can be extended to databases, both for compile time and run-time. A run-time information flow approach was investigated by Samarati et al. [9, 10]. A compile time approach for object-oriented databases was investigated by one of the authors in [8]. For this paper we do not distinguish between relational and object-oriented databases, but assume that all authorization information is associated with a *View*.

At this point we assume that a view has a single owner which can do the following:

1 grant access on the view to other users (read or write or define)

2 grant access with grant option. Users who get grant access can grant access to other users. Users who get both define and grant access can define new views based on their view and grant access to the new views to other users. A user who defines such a view becomes its owner.

In addition, we note the following:

- The view grant structure is known at compile time, and can be used by programs which use the view.
- The set of subjects who received read access on a view directly or indirectly is called the Closure of the view - $clos(V)$. Obviously if view V_1 is derived from view V_0 then $clos(V_0)$ contains $clos(V_1)$.

- Programs access the database by issuing a query on the view.

Since *clos*(*V*) is known at compile time, the query's label can be directly computed. However, since the query may not access all the attributes of the view, it is necessary to compute the closure of each accessible attribute separately. For an attribute *A*, we denote by *clos*(*A*, *V*) the closure of *A* with respect to view V and define it as $clos(A, V) = \{r : A \in V_j,$ V_j *derived from* V, r *can read* $V_j\}$. That is, the closure includes all the users granted read access for the relevant attributes via the original view, or via any view derived from it.

In order to define the label of *A* with respect to V, we need to consider the owners of the different views which derive V. Therefore, we define the label of *A* with respect to V as:
$label(A, V) = \{(o, r) : A \in V, V_j$ is *V or derives* $V, o = owner(V_j)$ *and* $r \in clos(A, V_j))\}$. This definition is justified as follows. Whenever we go up in the hierarchy, we are more restrictive, since we potentially add more owners, (so the basic view will have only one owner in the label). It is also justified by noting that *revocation* of rights in SQL databases is recursive.

To define the label of a *query* with respect to a view, we require that all the attributes that the query may access be defined in the view.

DEFINITION 15 (QUERY LABELS) *The label of a query Q which accesses attributes* $A_1, \ldots A_n$ *of a view V is:*
$label(Q, V) = \{(o, r) : \forall i = 1, \ldots, n\ A_i \in V_j$, V_j *is V or derived from V,* $o = owner(V_j)$ *and* $r \in INT(readers(label(A_i, V_j)))\}$.

This label is defined as the set of owners of each of the derived views that contain all attributes accessed by the query, and the set of all the readers who have access to *all* the attributes appearing in the query (for that view).

Figure 2 illustrates this definition. The figure shows a view V_1 with attributes *A, B, C, D,* and two views V_2 and V_3 derived from it, and the readers who have access to each view. Assume that the owner of V_1 and V_3 is o_1, and the owner of V_2 is o_2, then

- the label of a query accessing attributes *A, B* via the views V_2 and V_1 is $\{(o1 : r1, r2), (o2 : r2)\}$ and $\{(o1 : r1, r2)\}$, respectively (i.e. the query accessing V_2 has a more restrictive label.)
- the label of a query accessing attributes *C, D* via the views V_3 and V_1 is $\{o1 : r3\}$ and $\{o1 : r1, r3\}$ respectively.
- the label of a query accessing attributes *A, C* via the view V_1 is $\{o : r1\}$

The above discussion defines how to compute the label *label*(*Q, V*) for a query *Q* at compile-time. Using the above definitions one can define the notion of *safe flow* for databases similar to that of files, and apply both compile-time and run-time checks. We will not discuss it further here due to space limitations.

	A	B	C	D
V_3			r_3	r_3
V_2	r_2	r_2		
V_1	r_1	r_1	r_1	r_1

Figure 2. Example for Views labels.

5. CONCLUSIONS

Maintaining and enforcing the privacy and secrecy of data is recognized as a difficult problem, more so in an environment with decentralized control. Information flow control with the decentralized label model [6] is a step in the right direction. In the present paper, we have extended this work to deal in more details with files and I/O channels in section 3 and with databases and views in section 4. In the process, we have argued that the rule $L_2 \sqsubseteq L_1$ is too restrictive for many situations involving files and proposed more relaxed constraints that consider the readers to whom all the owners grant access.

Run-time checks are necessary to allow this form of controlled declassification. Though covert channels are possible, their impact can be limited by verifying the rights of the executors of a program.

Another issue is *Administration.* We have shown how various policies such as Delegation or Non-ORG control can be implemented using our model.

Among the several directions for future work is the investigation of alternative policies for declassification based, for example, on the presence of a given reader in the majority of the sets *readers*(L, o), or in all the sets *readers*(L, o) for a qualified subset of owners. Under investigation is also the problem of using the principal hierarchy to control the delegation, from an high level subject S_h to a lower level subject S_l. For example, if a supervisor S_h wants a technician S_l to run a backup program on his behalf, with our run-time policy the technician cannot, unless S_l is included in the label of the files. A useful extension of this policy would allow S_l to run the backup if S_l has an ancestor, with respect of the principal hierarchy, appearing as a reader in the label of the files.

Finally, we like to add that the definitions and proofs provided in this paper are quite intuitive. Our current research aims to develop a formal language and formal derivation rules on the labels, in the style of [6], to assist in the static analysis of the information flow.

Acknowledgments

This research was performed in part during a visit by the authors at the Center for Secure Information Systems at George Mason University, VA (USA).

References

[1] D. E. Bell and L. J. LaPadula. Secure Computer System: Unified Exposition and Multics Interpretation MTR-2997, MITRE Corp., Bedford, MA, March, 1976. reprinted in *J. of Computer Security* vol.4, no.2-3, pages 239–263, 1996.

[2] J. A. Goguen, and J. Meseguer. Security Policies and Security Models in *Proc. 1982 IEEE Symposium on Security and Privacy,* Oakland, CA, pages 11–20.

[3] J. McLean. Reasoning about Security Models, in *Proc. 1987 IEEE Symposium on Security and Privacy,* Oakland, CA, April 1987, pages 123–131. Also in *Advances in Computer System Security,* vol. III, ed. R. Turn, Artech House, Dedham, MA, 1988.

[4] A. C. Myers: JFlow: Practical Mostly-Static Information Flow Control. Proceedings, POPL 1999: 228-241

[5] A. C. Myers, and B. Liskov. Protecting Privacy using the Decentralized Label Model . *Trans. on Software Engineering and Methodology, vol.9, no.4,* October 2000, pages 410–442.

[6] A. C. Myers, and B. Liskov. Complete, Safe Information Flow with Decentralized Labels. in *Proc. IEEE Symposium on Security and Privacy,* Oakland, CA, May 1998, pages 186–197.

[7] R. Sandu and P. Samarati. Access Control: Principles and Practice. *IEEE Communication Magazine,* pages 40–48, 1994.

[8] M. Gendler and E. Gudes, "A compile-time Flow Analysis of Transactions and Methods in Object-oriented databases," Proceedings, 11 IFIP WG11.3 Database Security Conference, Lake Tahoe, CA. 1997.

[9] P. Samarati, E. Bertino, A. Ciampichetti and S. Jajodia "Information Flow Control in Object-Oriented Systems," *IEEE Trans. on Knowledge and Data Engineering,* July, 1997, 9(4), pages 524–539.

[10] P. Samarati, E. Bertino, A. Ciampichetti and S. Jajodia: Exception-Based Information Flow Control in Object-Oriented Systems. TISSEC 1(1), pages 26–65 (1998)

[11] S. Jajodia, R. Sandhu: Towards a Multilevel Secure Relational Data Model. Proceedings SIGMOD Conference 1991, pages 50–59

[12] S. Chen, D. Wijesekera, S. Jajodia: Flexflow: A flexible flow control policy specification framework Proceedings of IFIP WG11.3 Int. conference on data and application security, Estes park, Co., 2003.

[13] Yasuhiro Kirihata and Yoshiki Sameshima: A Web-based System for Prevention of Information Leakage proceedings of WWW2002, 2002.