Proceedings of the IEEE Symposium on Security and Privacy, May 1998, Oakland, California

Complete, Safe Information Flow with Decentralized Labels

Andrew C. Myers **Barbara Liskov**

MIT Laboratory for Computer Science
545 Technology Square, Cambridge, MA 02139
{andru,liskov}@lcs.mit.edu

Abstract

The growing use of mobile code in downloaded applications and servlets has increased interest in robust mechanisms for ensuring privacy and secrecy. Information flow control is intended to directly address privacy and secrecy concerns, but most information flow models are too restrictive to be widely used. The *decentralized label model* is a new information flow model that extends traditional models with per-principal information flow policies and also permits a safe form of declassification. This paper extends this new model further, making it more flexible and expressive. We define a new formal semantics for decentralized labels and a corresponding new rule for relabeling data that is both sound and complete. We also show that these extensions preserve the ability to statically check information flow.

1 Introduction

The growing use of mobile code in downloaded applications and servlets has increased interest in robust mechanisms for ensuring privacy and secrecy. A key problem is that information must be shared with downloaded code, while preventing that code from leaking the information. Information flow control is intended to address these privacy and secrecy concerns, but most information flow models are too restrictive to be widely used. This paper increases the power of a promising new model, the *decentralized label model* [ML97], making it more practical and useful.

Our goal is to check information flow by a straightforward static analysis of annotated program code. The idea is for a node to share information with a downloaded applet or uploaded servlet, yet prevent the mobile code from leaking the information; additionally, the applet or servlet could be protected from leaking its private information to other programs running on the same node.

The decentralized label model makes a good basis for information flow control because it improves on earlier models in several ways:

- It allows individual principals to attach flow policies to pieces of data. The flow policies of all principals are reflected in the *label* of the data, and the system guarantees that all the policies are obeyed simultaneously. Therefore, the model works even when the principals do not trust each other.
- The model allows individual principals to declassify labels by modifying their own flow policies. Arbitrary declassification is not possible because flow policies of other principals are still maintained. Declassification permits the programmer to remove restrictions when appropriate; for example, the programmer might determine that the amount of information being leaked is inconsequential. Previous work on information flow did not allow any declassifications within the model.
- It is compatible with static checking of information flow. Static analysis is required to prevent leakage of information through *implicit flows*, and to provide practical fine-grained control over information flow [DD77]. However, unless care is taken, static checking will be so restrictive as to make the model unusable. Our previous work [ML97] makes static analysis more expressive by supporting label polymorphism and safe run-time label checking. We have also demonstrated that label inference can be used to reduce the burden of adding static information flow annotations to a program.

This paper extends our previous work on decentralized labels to make the label system more flexible, while retaining the advantages we have just described. We make the following contributions here:

- We extend the model to allow safe relabelings that the previous work does not permit.
- We provide a formal definition of the model that allows us to define exactly what relabelings are legal. Our model

This research is supported by DARPA Contract F30602-96-C-0303, monitored by USAF Rome Laboratory. Andrew Myers is also supported by an Intel fellowship.
Web page: www.pmg.lcs.mit.edu

differs from earlier models [Den76, MMN90] because earlier approaches cannot deal with some safe relabelings that rely on relationships between different principals.

- We define a rule for static checking and prove that the rule is both sound and complete: it allows only safe relabelings, and it allows all safe relabelings.
- We also show that label checking and label inference can be done easily and efficiently using the new rule.

The rest of this paper is organized as follows. In Section 2, we briefly review the decentralized label model and show that it does not allow certain useful, intuitively safe relabelings. Section 3 provides a formal model for labels; it explains how a label can be interpreted as a set of *flows* and uses this to define legal relabelings. Section 4 sketches programming language annotations that permit static flow checking, presents the static relabeling rule and proves that it is both sound and complete, and shows that the relabeling rule can be used to statically analyze code with the annotations described. Related work is discussed in Section 5, and we conclude in Section 6.

2 Decentralized labels

This section provides a brief summary of the decentralized label model [ML97]. It also explains why its rules are too restrictive and what kinds of less restrictive rules are desirable.

2.1 Model

The decentralized label model is based on a notion of *labels* that allow individual owners of information to express their own policies. Owners are *principals*: identifiers representing users and other authority entities such as groups or roles. Some principals are authorized to *act for* other principals; this information is maintained in a *principal hierarchy* database. We assume that the principal hierarchy changes over time but that revocations occur infrequently. Also, at any moment, a process has the authority to act on behalf of some (possibly empty) set of principals.

Every value used or computed in a program execution has an associated label. A label L contains a set of owners, $\mathit{owners}(L)$; these are the principals whose data was observed in order to obtain that value. In addition, for each owner O, the label contains a set of readers, $\mathit{readers}(L, O)$; these are the principals that O allows to observe the value.

Observations happen when values are written to *output channels*. Each output channel C has an associated set of readers; these are the principals who will be able to observe information written to that channel (*e.g.*, the people that have access to a printer). A value can be written to a channel only if each reader of the channel has the authority to act for some reader in the *effective readers set* of the value's label. The effective readers set is the intersection of all the reader sets in the label. Restricting writing to channels like this ensures that each owner's policy is obeyed.

For example, for the label $L = \{o_1\colon r_1, r_2;\ o_2\colon r_2, r_3\}$ we have:

$$\begin{aligned} \mathit{owners}(L) &= \{o_1, o_2\} \\ \mathit{readers}(L, o_1) &= \{r_1, r_2\} \\ \mathit{readers}(L, o_2) &= \{r_2, r_3\} \\ \mathit{effectiveReaders}(L) &= \{r_2\} \end{aligned}$$

and a value labeled by L can be written to channel C provided all of C's readers can act for r_2.

In this model, every variable and input channel has a label. When a value is read from a variable or input channel, it acquires its label. When a value is written to a variable, the value's current label is forgotten; instead, it acquires the label of that variable. Therefore, assignment effectively creates a new copy of a value with a different label; to avoid information leaks, our rule requires that the new label must be the same as or more restrictive than the old one. (Changes in who can use information in a variable are accomplished by modifying the principal hierarchy.)

Assignment causes a *relabeling* of the value that is assigned. This kind of relabeling is termed a *restriction*. A relabeling is a restriction if the new label contains all the owners of the original, and the same or fewer readers for each owner. A restriction can be performed by any process, no matter what its authority. The expression $L_1 \sqsubseteq L_2$ means that L_1 is less restrictive than or equal to L_2, and that values can be relabeled from label L_1 to label L_2.

Values may also be relabeled by *declassification*, which reduces restrictiveness by removing an owner or adding a reader for an owner. Declassification can be performed only by a process with the authority to act for the owner whose policy is being changed; it requires a run-time check for the proper authority. The important point is that declassification cannot affect the policies of owners the process does not act for; since reading only occurs by the consensus of all owners, this limited declassification is safe.

Computations (such as multiplying two numbers) cause *joining* ($\sqcup$) of labels; the label of the result is the least restrictive label that reflects the policies in the labels of the values used in the computation:

$$\begin{aligned} \mathit{owners}(L_1 \sqcup L_2) &= \mathit{owners}(L_1) \cup \mathit{owners}(L_2) \\ \mathit{readers}(L_1 \sqcup L_2, O) &= \mathit{readers}(L_1, O) \cap \mathit{readers}(L_2, O) \end{aligned}$$

These rules follow from the definition of $\sqsubseteq$. Label inference also requires that the *meet* ($\sqcap$) of two labels be determinable; $A \sqcap B$ is the most restrictive label that can be be relabeled to both A and B. Its definition is dual to that of join.

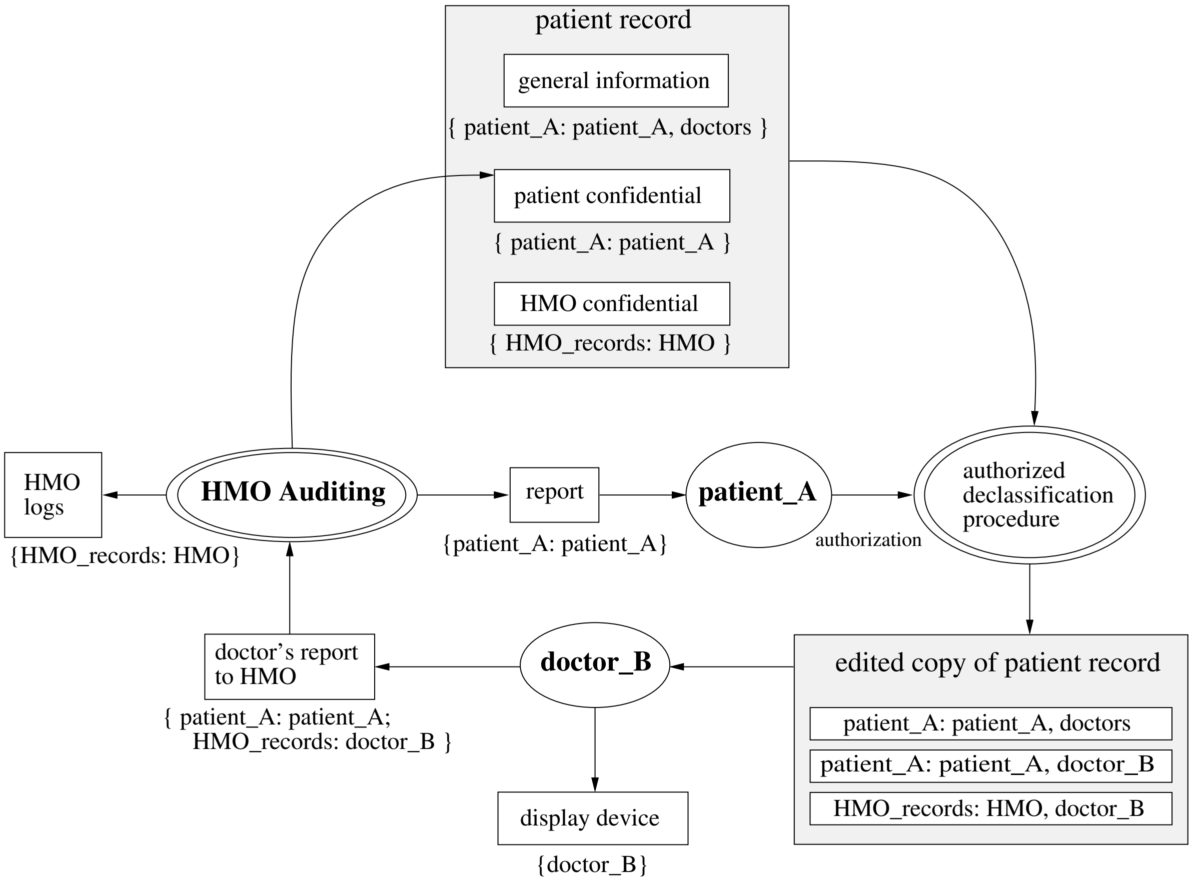

Figure 1: The patient/doctor example

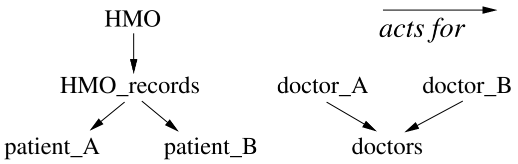

Figure 2: The patient/doctor principal hierarchy

2.2 Example

This section gives an example to illustrate the model. In the example, there are three parties with privacy concerns: a patient obtaining medical services, a doctor providing the services, and an HMO that serves as an intermediary. There are principals in the system for patients, *e.g.*, patient_A, and doctors, *e.g.*, doctor_B; all doctors can additionally act for doctors, which stands for the group of doctors within the HMO. Also, there are two HMO principals: HMO, representing maximum authority within the HMO, and HMO_records, representing authority over the record-keeping functions of the HMO; HMO can act for HMO_records, and HMO_records can act for patients: each patient must trust the HMO to keep track of its records. The resulting principal hierarchy is shown in Figure 2.

Figure 1 shows the system. The HMO maintains the patient's medical history; the example tracks information flow as the patient receives medical services. The patient record has three parts: general information about the patient, which is controlled by the patient but is readable by any doctor, private information (such as the medical history), which is normally not readable by doctors, and confidential information that the HMO does not release to patients.

The first step in a patient/doctor interaction is for the doctor to obtain a copy of the patient's record. The record is declassified so that the doctor can read it; this can only happen with the authorization of the patient. The patient makes an authenticated request to an existing program running with the authority of HMO_records; this program uses the patient's authority to provide the doctor with an edited version of patient's private information and of the HMO confidential information.

To read the information, the doctor requires an output channel to a display device with the single reader, doctor_B.

All information in the edited patient record can be written to such a channel, since doctor_B can act for doctors. The channel is created by code that authenticates doctor_B. Note that the patient information cannot be written to a channel that has any readers other than doctor_B, and that there is no way the doctor can declassify the patient information.

Eventually, the doctor sends a report to the HMO of services rendered. The report reflects all three components of the patient's record, so it acquires a joint label reflecting all these sources. Note that the joint label prevents the doctor from reading his own report, because the general patient information does not explicitly permit doctor_B as a reader. This is an example of unnecessary restrictiveness in the model.

The audit program runs with the authority of the HMO_records principal and thus can store the information with the appropriate labels in the log and the patient record database. It can also send a report to the patient; the designer of the audit program must use mechanisms outside the scope of information flow control to determine either that no HMO-confidential information is leaked or that the leak is acceptably small.

2.3 Limitations of the model

The rule for restrictions described earlier is not as general as we would like; it prevents us from doing valid relabelings that would simplify the example just presented. There are two kinds of such relabelings, both based on the existence of an acts-for relationship between principals.

- **Adding readers.** We should be able to add a reader r for some owner o if o already allows some reader r' that r acts for. This rule makes sense because allowing r' to read allows all principals that act for r' to read.
- **Replacing owners.** We should be able to replace an owner o' with some principal o that can act for o'. This rule makes sense because the new label only allows a process that acts for o to declassify it, while the original label allowed processes with weaker authority to declassify it.

If we allow adding readers, the doctor in the example is able to view his own report. The confidential patient information has the label {patient_A: patient_A,doctors}, which allows any doctor to view the data item, and therefore we ought to be able to relabel the item to explicitly allow a particular doctor to view it, *e.g.*, {patient_A: patient_A,doctor_B}. Since doctor_B is then a reader for every component of the joint label, he can view the report.

If we allow replacing owners, it has the advantage that the special rule of Section 2 is not needed for output channels; they can be treated as ordinary variables. Using the authority of the HMO, the display device can be assigned the label {HMO: doctor_B}. This labeling will allow all the information in the patient's record to be transmitted to the display device. The label {HMO: doctor_B} means that the HMO has certified that doctor_B is the only reader on this channel. There is no global notion of the readers of the channel; data owned by an owner o can only be written to this channel if o trusts the HMO (that is, HMO can act for o). The original trusted channels are easily modeled by assigning them the owner root (*i.e.*, a high-level principal).

Relabelings that add readers or replace owners can be done already, but only by a process with sufficient authority, using the declassification mechanism. Since the relabelings are restrictions, they ought not to require authority (although they do require a run-time check to determine whether a principal can act for another principal). We can use these relabelings to write useful procedures that run with minimal authority, observing the principle of least privilege [Sal74].

Providing these extensions also makes it easier to model desirable security policies. For example, suppose that a user wants to define security classes in a multi-level fashion: his own personal unclassified, classified, and secret classes for protecting his data. With these extensions, these three security classes can be represented as principals in the system, where the secret principal can act for classified, and classified for unclassified. The user can then assign security classes to other principals in the system by allowing them to act for one of these three principals; he correspondingly marks each data item as readable by the appropriate security class principal.

It is not trivial to extend the relabeling rule to permit these relabelings, because we want to preserve the ability to statically analyze information flow. As pointed out by Denning and Denning [DD77], information flow should be checked statically (*e.g.*, at compile time) to avoid leaks through *implicit flows*. The new relabelings above depend on the principal hierarchy as it exists at run time, and this structure cannot be known at compile time. So we need to be sure that any assumptions about the hierarchy that are used during compile time checking are valid for *all* hierarchies that might be encountered at run time.

We solve this problem in two steps. In Section 3 we give a formal model for labels that allows us to define legal relabelings. Then in Section 4 we define the rules for static checking and show that they are both sound and complete.

3 Extending and interpreting labels

The new relabelings depend on the existence of certain acts-for relationships, and therefore we need a rule that takes the principal hierarchy into account. In this section, we formalize the notions of labels and principal hierarchies and then define an intuitive condition for judging whether a relabeling rule is correct.

3.1 Generalizing labels

We will generalize the label model slightly, to allow an owner to be repeated within a label. (In Section 2, a label was characterized by an *owner set* in which each owning principal could only appear once, with its associated reader set.) As we will see later, allowing duplicate owners is important for maintaining the lattice structure of labels.

A label is a set of *components*, each of which expresses a *policy* for a single owner. The policy specifies a set of readers that are permitted by the owner to read the data. Different components of the label may have the same owner. The intuitive meaning of a label is that every component must be obeyed. If a component K is part of the label L ($K \in L$), then we will use the notation o_K to denote the owner of that component, and the notation R_K to denote the *set* of readers specified by that component. In the equations in this paper, the letters I, J, K will always denote label components.

3.2 Principal hierarchy

The principal hierarchy is defined by the acts-for relations between principals in the system. If x can act for y, we will denote this fact by the expression $x \succeq y$. The binary relation $\succeq$ is reflexive and transitive, but not anti-symmetric: two distinct principals may act for each other, in which case we say that the principals are equivalent. We use the notation $P \vdash x \succeq y$ to indicate that in the principal hierarchy P, the principal x can act for the principal y.

A principal hierarchy is a binary relation on principals, and can therefore be treated as a set of ordered pairs of principals that specifies all relations that exist. With this interpretation, $P \vdash x \succeq y$ is equivalent to $(x,y) \in P$. When one principal hierarchy P' contains more acts-for relations than another, P, we will say that P' *extends* P, which we will write as $P' \supseteq P$.

This model of principals is easily generalized. One obvious extension is to divide *acts-for* into more finely-grained (but transitive) authorizations. For example, all individual doctors might be able to *read* information for which doctors is an allowed reader, but they might not all be able to *declassify* information owned by that principal. This would help control the information if a doctor were ejected from the doctors group. Similarly, the ability to act for a principal p does not imply the ability to change who may act for p. We do not explore these extensions here for lack of space.

3.3 Interpreting labels

Intuitively, a relabeling is allowed if it does not create new ways for the relabeled information to flow. However, to specify this rule precisely, we need a simple way to *interpret* a label: that is, to decide what information flows are described by a label.

It is useful to think of a label as describing a set of *flows*, where a flow is an (owner, reader) pair. If a label L has a component K with owner o_K, then it describes flows (o_K, r) for every reader r in the set R_K. If a principal o' is not an owner in the label, L describes flows (o', r) for every principal r. Intuitively, this means that o' has not expressed a flow policy for the labeled data, so it permits flows to any principal.

Under the interpretation of labels as sets of flows, the earlier relabeling rules described in Section 2 can be expressed quite simply. Relabeling is permitted from L_1 to L_2 (*i.e.*, $L_1 \sqsubseteq L_2$) exactly when $L_1 \supseteq L_2$ — when L_2 is at least as restrictive as L_1. In other words, the partial order on labels is exactly the partial order on sets of flows. For this reason, we call the relabeling rule of Section 2 the *subset relabeling rule*. Similarly, the join of two labels, $L_1 \sqcup L_2$, is simply their intersection, $L_1 \cap L_2$. The meet of two labels, $L_1 \sqcap L_2$, is the same as the union of the labels, $L_1 \cup L_2$.

3.4 Flow set constraints

The subset relabeling rule is too restrictive because it does not take the principal hierarchy into account. By thinking about the label as a set of flows, we will observe that there are two constraints that a set of flows ought to satisfy in a particular principal hierarchy — one constraint on readers, and one on owners. We will use these constraints to construct a less restrictive relabeling rule.

The *reader constraint* is as follows. If a set of flows contains a flow (o, r), and r' is a principal that can act for r, then the set must also contain the flow (o, r'). For example, the label {patient_A: doctors} is equivalent to the label {patient_A: doctors, doctor_B}, since the principal doctor_B can act for the principal doctors.

The idea here is that although a label explicitly states some set of flows, the actual flows denoted by the label depend on the principal hierarchy. We call the set of denoted flows the label's *interpretation* in the principal hierarchy. We will define a function $\mathbf{X}$ that maps a label to its interpretation. Using the definition of $\mathbf{X}$, all of the intuitively sound relabelings described in Section 2 are easily described. The function $\mathbf{X}$ takes the current principal hierarchy as a (for now) implicit argument. Thus, the label $\mathbf{X}L$ represents the interpretation of label L in the current principal hierarchy. The reader constraint just described can be stated more formally as follows:

$$r' \succeq r \ \& \ (o, r) \in \mathbf{X}L \rightarrow (o, r') \in \mathbf{X}L$$

However, the reader constraint is not sufficient, because we also want to allow relabelings that change the label's owners. Consider the relabeling from {patient_A: doctor_B} to {HMO_records: doctor_B}. This relabeling effectively transfers the responsibility of controlling the flow of the data from the principal patient_A to the principal HMO_records. This transfer restricts the data's flow, since HMO_records can act for patient_A. The key insight to allowing this kind of relabeling is an *owner constraint*:

$$o' \succeq o \ \& \ (o,r) \in \mathbf{X}L \rightarrow (o',r) \in \mathbf{X}L$$

The symmetry of this rule to the reader constraint might seem incorrectly to imply that the inferior principal o can dictate the addition of readers to the reader set of o'. The interpretation is different: when a superior owner states that a flow must not occur, this flow is removed from the reader sets of all inferior owners. However, if a superior owner does not try to prevent a flow, inferior owners may still prevent it. Thus, the inferior owner's policy must be at least as restrictive as the superior owner's policy. The owner constraint can be written in an equivalent, negative form that captures this intention more directly:

$$o' \succeq o \ \& \ (o',r) \notin \mathbf{X}L \rightarrow (o,r) \notin \mathbf{X}L$$

Using this constraint, the label {HMO_records: doctor_B} is seen to be equivalent to the label {HMO_records: doctor_B; patient_A: doctor_B}, in the principal hierarchy of Figure 2. While the first label would seem to allow flows from patient_A to all readers, the owner constraint prevents the reader set of patient_A from being larger than that of HMO_records.

3.5 Label functions

To help construct the label interpretation function $\mathbf{X}$, we define two functions that establish the reader and owner constraints. First, we define a function $\mathbf{R}$ that expands a set of readers to include the implicitly allowed readers described by the reader constraint. It adds to the readers R_I of a component I to produce an expanded reader set $\mathbf{R}R_I$:

$$\mathbf{R}R_I = \{r \mid \exists_{r' \in R_I} : r \succeq r'\}$$

We also define a function $\mathbf{O}$ that converts a label into a set of flows by restricting it so that it obeys the owner constraint. Its form is roughly dual to that of $\mathbf{R}$:

$$\mathbf{O}L = \{(o,r) \mid \forall_{I \in L} : o_I \succeq o \rightarrow r \in R_I\}$$

As we would expect, both $\mathbf{R}$ and $\mathbf{O}$ are monotonic in the set or label they manipulate, in the sense that if $R_1 \supseteq R_2$, then $\mathbf{R}R_1 \supseteq \mathbf{R}R_2$ and if $L_1 \supseteq L_2$, then $\mathbf{O}L_1 \supseteq \mathbf{O}L_2$. However, the two functions differ in their behavior as the principal hierarchy changes. Making the principal hierarchy P an explicit argument to the functions, we have the following: if the principal hierarchy P' is an extension of P ($P' \supseteq P$), then $\mathbf{R}(R,P') \supseteq \mathbf{R}(R,P)$, but $\mathbf{O}(L,P') \subseteq \mathbf{O}(L,P)$: $\mathbf{O}$ is anti-monotonic in the P argument.

By composing the $\mathbf{R}$ and $\mathbf{O}$ functions, we obtain the label interpretation function $\mathbf{X}$, which maps a label to a set of flows, given a particular principal hierarchy:

$$\mathbf{X}L = \{(o,r) \mid \forall_{I \in L} : o_I \succeq o \rightarrow r \in \mathbf{R}R_I\}$$

The result of $\mathbf{X}L$ satisfies both the reader and owner constraints, since $\mathbf{O}$ preserves the reader constraint established in each component by $\mathbf{R}$. Intuitively, the effect of applying both $\mathbf{R}$ and $\mathbf{O}$ is the following: a flow (o,r) is implied by a label L if every owner who can act for o permits the flow — either explicitly, by allowing r to read it, or implicitly, by allowing some principal that r can act for to read it.

Using the function $\mathbf{X}$, we can now write the *correctness condition* for relabeling in the presence of an arbitrary principal hierarchy. The relabeling from L_1 to L_2 in principal hierarchy P is valid as long as no new flows are added. Making the principal hierarchy an explicit argument to $\mathbf{X}$, the correctness condition is the following:

$$\mathbf{X}(L_1,P) \supseteq \mathbf{X}(L_2,P)$$

We can apply this rule to show that the relabeling from L_1 = {patient_A: doctors} to L_2 = {HMO_records: doctor_B} is valid. Applying $\mathbf{X}$ to L_2 gives us a set containing the flow (HMO_records, doctor_B) and the flows (p, doctor_B) for every patient p (since HMO acts for all patients), as well as other flows (o, r) for unrelated owners o and all readers r. Applying $\mathbf{X}$ to L_1 gives us a set containing all these pairs and more: (HMO_records, r) for every r, for example. Because $\mathbf{X}L_1 \supseteq \mathbf{X}L_2$, the relabeling from L_1 to L_2 is legal.

Because the function $\mathbf{X}$ is a composition of $\mathbf{R}$ and $\mathbf{O}$, it is monotonic with respect to L, but neither monotonic nor anti-monotonic with respect to P. It also has some other interesting properties. We can interpret the set produced by applying $\mathbf{X}$ to a label as a label itself (although one that is probably too large to write down!); this is the label in which every flow is mentioned explicitly. With this interpretation, we can see that like $\mathbf{O}$ and $\mathbf{R}$, the function $\mathbf{X}$ is idempotent; that is, $\mathbf{X}L = \mathbf{X}\mathbf{X}L$.

The function $\mathbf{X}$ can also be thought of as a closure operator that converts a label to a closed set of flows. In accordance with this interpretation, the set of labels produced by $\mathbf{X}$ is closed under intersection and union of labels.

4 Checking relabeling statically

We wish to support static checking of programs containing label annotations, because static checking allows precise, fine-grained analysis of information flows, and can capture implicit flows properly [DD77], whereas dynamic label checks create information channels that must be controlled through additional static checking [ML97]. However, the correctness condition ($\mathbf{X}L_1 \supseteq \mathbf{X}L_2$) derived in Section 3 cannot be used directly in static checking since it depends on the principal hierarchy at the time that the relabeling takes place, while static checking is done earlier, perhaps as part of compilation. The principal hierarchy may have changed between checking and execution, so the full run-time principal hierarchy is not available when relabeling is checked. Therefore, relabeling must be checked using only partial information about the principal hierarchy.

In this section, we develop a general rule for checking relabelings statically using partial information about the principal hierarchy. We begin by giving a sketch of how programs are annotated. Then we define the relabeling rule and show that it is both sound and complete. Then we discuss the practicality of the system, arguing that both label checking and label inference are practical.

4.1 Annotations

We assume that programs are statically annotated with information about the labels of data that they manipulate, and that programs are checked by a static label checker that statically analyzes information flows to determine whether the program follows the information flow rules.

In [ML97], a set of language annotations is described that permits static information-flow checking. Here we summarize the important features to give an idea of the framework, and describe new annotations needed to support the extended relabeling rule.

- All variables, arguments, and procedure return values have labeled types. For example, a labeled integer variable might be declared as int{patient_A: doctors} x;. The label may be omitted from a local variable, causing it to be inferred automatically. If the label is omitted from a procedure argument, it is an implicit parameter, and the procedure is generic with respect to it.
- The actsFor statement allows a run-time test of whether the process running the code can act for a principal. In actsfor (p) S, the statement S is executed only if the process can act for principal p; the label checker will allow declassifications on behalf of p within S.
- The expression declassify(e, L) relabels the value e with the label L. The label L may add readers to the label of e for some owners o_i, or remove some owners o_i; the statement is legal only if a containing actsFor statement has established that the process can act for each of the o_i.
- Procedures are assigned a principal when they are compiled; this principal derives from the user who is running the compilation. When a procedure is called it always runs under this authority. Callers can additionally grant the called procedure the authority to act for principals they act for (recall that a process may act on behalf of several principals), but this must be done explicitly.
- Variables and arguments may be declared to have the special base type label, which permits run-time label checking. Variables of type label and argument-label parameters may be used to label variables that are mentioned within the procedure body. They also may be used in declassify expressions.
- A labelcase statement can be used to determine the run-time label of a value, and a special type protected conveniently encapsulates a value along with its run-time label.

The following extensions to this previous framework enable static reasoning about the principal hierarchy:

- Variables of the special type principal may also be used in labels and in actsFor statements. Also, when a procedure is granted the authority of some principal by its caller, the identity of the principal is placed in an argument of type principal.
- A second kind of actsFor statement: In actsFor(p_1, p_2) S, the statement S is executed only if a run-time test determines that principal p_1 can act for principal p_2. The label checker then uses the knowledge that $p_1 \succeq p_2$ when checking relabelings that occur within S.

For example, using the actsFor extension, in

```
int{patient: doctors} x;
int{patient: doctor_B} y;
actsFor (doctor_B, doctors) y = x;
```

the assignment is legal because within the body of the actsFor statement the checker knows that doctor_B can act for doctors.

For each program statement that the label checker verifies, some acts-for relations can be determined to exist, based on the lexical nesting of the actsFor statements. These relations form a subset of the true principal hierarchy that exists at run time; all that is known statically is that the true principal hierarchy contains the explicitly stated acts-for relations.

Using this fairly general model for programming with static information flow annotations, the challenge is to define a sound (conservative) rule for checking relabelings. In the next section, we show that defining such a rule is not as simple as one might expect. We then present a rule that is not only sound but also complete, in that it permits every relabeling that cannot be used to leak information.

4.2 Static correctness condition

When a program assigns a value to a variable, it relabels the data being assigned, since the value's label is changed to be the same as the label on the variable. This relabeling is sound as long as it does not create new ways for the assigned data to flow. One example of a sound relabeling rule is the original subset relabeling rule; if $L_1 \supseteq L_2$ (L_1 is the value's label and L_2 is the variable's label), the monotonicity of $\mathbf{X}$ guarantees that the correctness condition holds, regardless of the principal hierarchy. However, the subset relabeling rule, as we've seen, is excessively restrictive. We would like a rule that recognizes the principal hierarchy.

Let P be a principal hierarchy that contains only the acts-for relations that are statically known based on the containing actsFor statements. We will refer to this principal hierarchy as the *static principal hierarchy*. The actual principal hierarchy at run time is an extension of P; it must contain all of the acts-for relations in P, plus possibly additional relations. If P' is the actual principal hierarchy, we have $P' \supseteq P$. Using this notation, and introducing the principal hierarchy as an explicit argument to the function $\mathbf{X}$, we can express the *static correctness condition*: it is safe to relabel from L_1 to L_2 in P if the following holds:

$$\forall_{P' \supseteq P} : \mathbf{X}(L_1, P') \supseteq \mathbf{X}(L_2, P')$$

One might expect that to check whether a relabeling is valid, we should check whether $\mathbf{X}(L_1, P) \supseteq \mathbf{X}(L_2, P)$, *i.e.* apply the correctness condition for the principal hierarchy P. By construction, this rule allows all valid relabelings to take place; if a relabeling is not allowed by this rule, then it creates new flows in the principal hierarchy P. However, the following example will show that this rule is not sound.

Consider the following (bad) relabeling from L_1 to L_2

L_1 = { doctors: patient_A; doctor_B: patient_A, patient_B }
L_2 = { doctors: doctors, patient_A ;
doctor_B: patient_A, patient_B }

Now, consider what happens when we apply $\mathbf{X}$ to both of these labels while assuming that the principal hierarchy P' contains the single relation doctor_B $\succeq$ doctors:

$\mathbf{X}L_1$ = { doctors: patient_A; doctor_B: patient_A, patient_B }
$\mathbf{X}L_2$ = { doctors: patient_A; doctor_B: patient_A, patient_B }

Note that $\mathbf{X}L_2$ does not contain the flow (doctors, doctors) because the superior owner doctor_B rules it out. Since these two label interpretations are equal, it would seem that the relabeling is correct. However, if we learn that patient_B is also a doctor (patient_B $\succeq$ doctors), applying $\mathbf{X}$ to both labels leads to a quite different conclusion:

$\mathbf{X}L_1$ = { doctors: patient_A; doctor_B: patient_A, patient_B }
$\mathbf{X}L_2$ = { doctors: patient_B, patient_A;
doctor_B: patient_A, patient_B }

The relabeling is invalid under the principal hierarchy P', because it adds the flow (doctors, patient_B). This example shows that the correctness condition cannot be applied directly as a relabeling rule.

4.3 A sound and complete relabeling rule

The correct rule for checking a relabeling from label L_1 to label L_2 is intuitive: for every component I in L_1, there must be a corresponding component J in L_2 that is at least as restrictive as I. The component J is at least as restrictive as I if $o_J \succeq o_I$ and $R_J \subseteq \mathbf{R}R_I$. If L_1 can be relabeled to L_2 under principal hierarchy P, we will write $P \vdash L_1 \sqsubseteq L_2$. This condition is defined formally as follows:

$$\forall_{I \in L_1} \exists_{J \in L_2} \left[P \vdash o_J \succeq o_I \ \&\ R_J \subseteq \mathbf{R}(R_I, P) \right]$$

Expanding the definition of $\mathbf{R}$, we obtain the following equivalent and more symmetrical formulation:

$$\forall_{I \in L_1} \exists_{J \in L_2} \left[P \vdash o_J \succeq o_I \ \&\ \forall_{r_j \in R_J} \exists_{r_i \in R_I} : P \vdash r_j \succeq r_i \right]$$

The binary relation $\sqsubseteq$ is defined for any principal hierarchy P. The relation is a *pre-order*: it is transitive and reflexive, but not anti-symmetric, since two labels may be equivalent without being equal. If A and B are equivalent, we write $A \approx B$ to mean $A \sqsubseteq B \ \&\ B \sqsubseteq A$. For example, with the hierarchy of Figure 2, the labels {HMO: doctors} and {HMO: doctors, doctor_A} are equivalent. Every principal hierarchy generates a pre-order on labels, defining the legal relabelings.

The nature of the relabeling rule can be understood by considering the incremental relabelings that it permits. It allows an arbitrary sequence of any of the following four kinds of relabelings, each of which is clearly sound individually:

- A reader may be dropped from some owner's reader set.
- A new owner may be added to the label, with an arbitrary reader set.
- A reader may be added as long as it can act for some member of the reader set.
- An owner may be replaced by an owner that acts for it.

Interestingly, these incremental relabelings also capture *all* of the sound relabelings. That is, the rule for $\sqsubseteq$, which we will call the *complete relabeling rule*, is both sound and complete. When we say that the rule is complete, we mean that it exactly captures the set of valid relabelings, with respect to the static correctness condition defined in Section 4.2, and using our assumptions about the static checking environment. We now provide sketches of our formal proofs for these claims. (The rule has also been checked for soundness using Nitpick, a counterexample generator [JD96].)

Soundness. We must show that if the relabeling rule holds for some principal hierarchy P, the correctness condition holds for all possible extensions P':

$$(P \vdash L_1 \sqsubseteq L_2) \rightarrow [\forall_{P' \supseteq P} : \mathbf{X}(L_1, P') \supseteq \mathbf{X}(L_2, P')]$$

Suppose that L_1 can be relabeled to L_2, $P' \supseteq P$, and $\mathbf{X}(L_1, P')$ does not contain some flow (o, r). We will show that (o, r) cannot be in $\mathbf{X}(L_2, P')$ either. If (o, r) is not in $\mathbf{X}(L_1, P')$, there must be some owner o_I in L_1 that suppresses it (*i.e.*, $r \notin \mathbf{R}(R_I, P')$ and $P' \vdash o_I \succeq o$). Since $P \vdash L_1 \sqsubseteq L_2$, there is a corresponding owner o_J in L_2 such that $R_J \subseteq \mathbf{R}(R_I, P)$ and $P \vdash o_J \succeq o_I$. Since $P \vdash o_J \succeq o_I$, we have $P' \vdash o_J \succeq o_I$, and transitively $P' \vdash o_J \succeq o$. We now show that this o_J prevents (o, r) from appearing in $X(L_2, P')$.

Let r' be an arbitrary reader such that $P' \vdash r \succeq r'$. We know that $r' \notin R_J$. (If $r' \in R_J$, we would have a contradiction: $r' \in \mathbf{R}(R_I, P)$, so $r' \in \mathbf{R}(R_I, P')$, and therefore $r \in \mathbf{R}(R_I, P')$.) Since for all such r', $r' \notin R_J$, we have $r \notin \mathbf{R}(R_J, P')$. Since we also know $P' \vdash o_J \succeq o$, this means $(o, r) \notin \mathbf{X}(L_2, P')$. Since this was true for arbitrary o and r, any flow not in $\mathbf{X}(L_1, P')$ is also not in $\mathbf{X}(L_2, P')$. Therefore, the relabeling rule is sound.

Completeness. We must show the converse:

$$\left[\forall_{P' \supseteq P} : \mathbf{X}(L_1, P') \supseteq \mathbf{X}(L_2, P')\right] \rightarrow (P \vdash L_1 \sqsubseteq L_2)$$

We prove this statement by contradiction: if a relabeling is rejected by the rule ($L_1 \not\sqsubseteq L_2$), we can find a P' such that $P' \supseteq P$ but $\mathbf{X}(L_1, P') \not\supseteq \mathbf{X}(L_2, P')$. In other words, if a relabeling is rejected, it might result in a leak.

If $\neg(P \vdash L_1 \sqsubseteq L_2)$, there must be some owner o_I in L_1 such that for every component J in L_2 where $o_J \succeq o_I$, $R_J \not\subseteq \mathbf{R}R_I$. Consider an arbitrary such component J in L_2 (if there is no such J, the relabeling leaks even in P). The component J must have a reader r_j where $r_j \in R_J$ but $r_j \notin \mathbf{R}R_I$. We will now use the readers r_j of every such J to construct a principal hierarchy P' that extends P and results in a leak.

Consider a principal hierarchy P' that is exactly like P, except that there is an additional principal r that in P is unrelated to any of the owners or readers in L_1 and L_2. We form P' by adding a relation (r, r_j) for each r_j and taking the transitive closure:

$$P' = P \cup \{(r, r') \mid \exists r_j : (r_j, r') \in P\}$$

Using this definition, we find that $(o_I, r) \in \mathbf{X}(L_2, P')$ but $(o_I, r) \notin \mathbf{X}(L_1, P')$, which shows that the relabeling causes a leak in P'. Therefore, the relabeling rule is complete.

This completeness result can be strengthened further: our rule is complete even in the presence of negative information about the principal hierarchy. We could imagine acquiring negative information by allowing an `else` clause in the `actsFor` statement. Since `actsFor` tests whether one principal can act for another, in the body of the `else` clause we would be able to determine statically that the specified principal relationship does *not* exist. This static information could be used to establish an *upper bound* on the dynamic principal hierarchy. However, an upper bound is not useful in checking relabelings: the proof for completeness still holds in the presence of an upper bound on P', since we can simply choose an arbitrary r that is not mentioned in the upper bound.

4.4 Static checking

Now we consider what is involved in doing static checking. We have already explained how to check assignments: we use the complete relabeling rule. But the labels being compared may be the results of joins (to account for computations), and meets (if the checker is doing label inference). Therefore, we need to define join and meet.

Labels form a pre-order rather than a lattice or even a partial order, because two labels can be equivalent without being equal. However, labels do preserve the important properties of a lattice that make static reasoning about information flow feasible: any pair of elements possesses least upper bounds and greatest lower bounds, which are unique to within an equivalence class. In addition, the join and meet operations distribute over each other.

Below we define join and meet. Our definitions have the desirable properties that they are easy to evaluate and that the resulting labels are easy to deal with when applying the complete relabeling rule.

4.4.1 Join. The join, or least upper bound, is useful in assigning a label to the result of an operation that combines several values, such as adding two numbers. The result of adding two numbers ought in general to be restricted at least as much as the numbers being added. However, we would also like not to restrict the sum unnecessarily; therefore, it is assigned the *least* restrictive label that is no less restrictive than both input labels. In a lattice, there is a unique least label; however, uniqueness is not important for our purposes. Any label within an equivalence class is acceptable as long as it can be relabeled to every label that is at least as restrictive as the input labels.

The join of two label expressions can be defined quite simply: it is the concatenation of all their components. The following are examples of join expressions, where A, B, and C are principals unrelated by the acts-for relation:

$$\{A : B\} \sqcup \{B : C\} = \{A : B; B : C\} \quad (1)$$
$$\{A : B\} \sqcup \{A : B, C\} = \{A : B\} \quad (2)$$
$$\{A : B\} \sqcup \{A : C\} = \{A : B; A : C\} \quad (3)$$

After doing a join, the compiler can sometimes simplify the label expression by removing redundant components, so that future checking steps run more efficiently. This simplification has been performed in the second example. A component is redundant if the relabeling rules behave identically for the label regardless of whether the component is present. One component $o_I : R_I$ makes another component $o_J : R_J$ redundant if $o_I \succeq o_J$ and $R_I \subseteq \mathbf{R}R_J$. In all possible relabelings involving such a label, the presence of component J will not affect the validity of a relabeling.

The third example illustrates the difference between this join operator and the earlier one defined in Section 2, based on the subset relabeling rule. The earlier join definition results in the label $\{A : \emptyset\}$, since reader sets for the same owner are intersected. The difference between the two join results may seem inconsequential; however, if $C \succeq B$; then the label $\{A\colon B; A\colon C\}$ can be relabeled to the label $\{A\colon C\}$, but $\{A\colon \emptyset\}$ cannot. Therefore, the difference in the rules is significant.

We can now see why it is important that owners be repeatable in labels: it completes the lattice of equivalence classes. If repeated owners were not allowed, there would be no least upper bound for many pairs of labels. Consider the third example again, but disallowing repeated owners. If A' is another principal with $A' \succeq A$, then the least restrictive labels that both $\{A\colon B\}$ and $\{A\colon C\}$ could be relabeled to would include $\{A\colon \emptyset\}$, $\{A\colon B; A'\colon C\}$, and $\{A'\colon B; A\colon C\}$, none of which can be relabeled to any other. There would be no least upper bound for these two labels.

The join operation just described produces the least upper bound of two labels. This can be seen by interpreting a join result as a set of flows, in an extended principal hierarchy P'. It follows directly from the definition of $\mathbf{X}$ that for all such hierarchies P',

$$\mathbf{X}(A \sqcup B, P') = \mathbf{X}(A, P') \cap \mathbf{X}(B, P')$$

This equation means that there is no label less restrictive than $A \sqcup B$ that both A and B can be relabeled to. Therefore, the join operator produces the least upper bound of the two labels, to within an equivalence class.

4.4.2 Reasoning about joins.

Components of a join can be independently relabeled or declassified. The property is important because it allows checking of code that is generic with respect to some of the labels that appear in it. In the case of declassification, there are no surprises for the declassifying principal: the set of flows that are added by declassifying a join is always a subset of the set of flows that would be added by declassifying the individual components. There are no interactions between the two parts of the join that create new, unexpected flows.

For example, if label L_1 can be relabeled to L_2, then $L_1 \sqcup L_3$ can be relabeled to $L_2 \sqcup L_3$, regardless of what L_3 is. L_3 may be an opaque label, or even a label that is determined at run time, without invalidating the relabeling. Similarly, if L_1 can be declassified to L_2, then $L_1 \sqcup L_3$ can be safely declassified to $L_2 \sqcup L_3$. These relabelings and declassifications work because the join guarantees that all components in L_3 will be respected.

4.4.3 Meet.

The meet or greatest lower bound of two labels is the most restrictive label that can be relabeled to both of them. The meet of two labels is not produced by computations during the program's execution, but it is useful in defining algorithms for automatic label inference [DD77, ML97]. The meet is useful to automatically infer labels for inputs, just as the join is useful to produce labels for outputs. For example, in the following code, the most restrictive label x could have can be expressed by using a meet:

```
int x;
int{A} y;
int{B} z;
y = x;
z = x;
```

In this example, the variables y and z have labels of A and B respectively. The variable x can be assigned any label C so long as it can be relabeled to both A and B. Therefore, $A \sqcap B$ is an upper bound on the label for x. The simple algorithm for inferring variable labels that we have described elsewhere [ML97] uses a succession of meet operations in this fashion to refine unknown variable labels downward until either all variables have consistent assignments, or a contradiction is reached.

To construct the meet of two labels, let us first consider the meet of two components J and K. If there is no statically known relation between the owners of these components, the meet is $\{\}$. This is the result obtained when either J or K is uninterpreted (*e.g.*, is a label parameter), or when both have known owners but there is no static relationship established between them (by some containing actsFor statement). Otherwise, suppose that $J = \{o : r_1 \ldots r_n\}$ and $K = \{o' : r'_1 \ldots r'_n\}$. if o' can act for o (which includes the case where they are equal), then the meet of the two components is $\{o :\ r_1 \ldots r_n, r'_1 \ldots r'_n\}$.

Now, consider the meet of two arbitrary labels. Since a label containing several components is really the join of these components, the meet can be computed by distributing the meet over both joins. The result of the meet is the join of all pairwise meets of components, using one component from each label. Some of these pairwise meets may produce the label $\{\}$, which of course can be dropped from the join.

As with the formula for join, the validity of this formula for meet can be seen by using the label interpretation function. If P' is some extension of the principal hierarchy used to compute the meet of labels A and B, then

$$\mathbf{X}(A \sqcap B, P') \supseteq \mathbf{X}(A, P') \cup \mathbf{X}(B, P')$$

The formula for meet is sound, but unlike the formula for join, it does not always produce the most restrictive label for all possible extensions P'. This happens because the rule for joining two components must return $\{\}$ when the owners are not known to have a relationship, though in the real hierarchy, a relationship may exist. The result is that label inference must be conservative in some cases, which does not seem to be a significant problem since even explicit label declarations do not work in those cases.

It can also be shown straightforwardly that join and meet distribute over each other in the expected way, producing equivalent labels:

$$\begin{aligned} A \sqcup (B \sqcap C) &\approx (A \sqcup B) \sqcap (A \sqcup C) \\ A \sqcap (B \sqcup C) &\approx (A \sqcap B) \sqcup (A \sqcap C) \end{aligned}$$

This means that a static checker doing label inference as described elsewhere [ML97] can rely on the properties of meet and join to simplify label expressions.

5 Related work

There has been much work on information flow control and on the static analysis of security guarantees. The lattice model of information flow comes from the early work of Bell and LaPadula[BL75] and Denning [Den76].

The decentralized label model has several similarities to the ORAC model [MMN90]: both models provide some approximation of the "originator-controlled release" labeling used by the U.S. DoD/Intelligence community. Both also support the joining of labels as computation occurs, though the ORAC model lacks some important lattice properties. Unlike our model, ORAC is intended to be dynamically checked. Dynamic checks result in storage and run-time overhead, and data can become more and more stringently labeled as it is used. Further, the label checks themselves can become a covert channel. We have shown that static checking can be used to control this channel [ML97]. Interestingly, ORAC does allow owners to be replaced in label components (based on ACL checks that are analogous to actsFor checks); however, it does not support declassification or the new relabelings described in this paper.

Other work on information flow policies has examined complex aggregation policies for commercial applications [CW87, BN89, Fol91]. We have not addressed policies that capture conflicts of interest, though our fine-grained tracking of ownership information seems applicable. Many of these information control models are not designed to be checked statically. IX [MR92] is a good example of a real-world information flow control system that used dynamic checking. Recent work by Ferrari et. al [FSBJ97] introduces a form of dynamically-checked declassification through special *waivers* to strict flow checking. Some of the need for declassification in their framework would be avoided with fine-grained static analysis. Because waivers are applied dynamically and mention specific data objects, they seem likely to have administrative and run-time overheads.

Static analysis of security guarantees also has a long history. It has been applied to information flow [DD77, AR80] and to access control [JL78, RSC92]. There has recently been more interest in provably-secure programming languages, treating information flow checks in the domain of type checking. Some of this work has focused on formally characterizing existing information flow and integrity models [PO95, VSI96, Vol97]. Smith and Volpano have recently examined the difficulty of statically checking information flow in a multithreaded environment [SV98]; we have not addressed this problem. Heintze and Riecke [HR98] have shown that information-flow-like labels can be used in a simple functional language to statically check an integrated model of access control, information flow control, and integrity. Their model does not, however, allow declassification of information flows or run-time flow checking. Also, Abadi [Aba97] has examined the problem of achieving secrecy in security protocols, also using typing rules, and has shown that encryption can be treated as a form of safe declassification through a primitive encryption operator.

6 Conclusions

The decentralized label model is a promising approach for making information flow a practical way to guarantee secrecy and privacy. It provides considerable flexibility by allowing individual principals to attach flow policies to individual values manipulated by a program. These more flexible labels then permit values to be declassified by an owner of the value. This declassification is safe because it does not affect the secrecy guarantees to other principals who have an interest in the secrecy of the data. This support for multiple principals makes the label model ideal for mutually distrusting principals.

However, while the original model contained a principal hierarchy, the hierarchy was not fully integrated into the relabeling rules, making the rules unnecessarily restrictive. This paper has defined a complete relabeling rule for the decentralized label model. The new rule precisely captures all the legal relabelings that are allowed when knowledge about the principal hierarchy is available statically. We have shown that the rule is both sound and complete, and furthermore that it is easy to apply. We have formalized the relabeling rule as a pre-order relation with distributive lattice properties: join and meet operators can be defined on these labels, which means that a compiler or static checker can use them to check information flow statically, to support label polymorphism, and to do label inference.

The new rules for relabeling, join, and meet make the decentralized label model more practical and more usable. They make it easier to model common security paradigms, allowing control of information flow in a system with group or role principals. They also allow individual principals to model their own multilevel security classes conveniently.

Acknowledgments

The authors would like to acknowledge the many helpful comments they have received about this work, including suggestions from Martín Abadi, Kavita Bala, Phillip Bogle, Chandrasekhar Boyapati, Miguel Castro, Stephen Garland, Jason Hunter, and the reviewers. We would also like to thank Daniel Jackson for his help with Nitpick.

References

[Aba97] Martín Abadi. Secrecy by typing in security protocols. In *Proc. Theoretical Aspects of Computer Software: Third International Conference*, September 1997.

[AR80] Gregory R. Andrews and Richard P. Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems*, 2(1):56–76, 1980.

[BL75] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, MITRE Corp. MTR-2997, Bedford, MA, 1975. Available as NTIS AD-A023 588.

[BN89] D. F. Brewer and J. Nash. The Chinese wall security policy. In *Proc. IEEE Symposium on Security and Privacy*, pages 206–258, May 1989.

[CW87] David Clark and David R. Wilson. A comparison of commerical and military computer security policies. In *Proc. IEEE Symposium on Security and Privacy*, pages 184–194, 1987.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, 1977.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, 1976.

[Fol91] Simon N. Foley. A taxonomy for information flow policies and models. In *Proc. IEEE Symposium on Security and Privacy*, pages 98–108, 1991.

[FSBJ97] Elena Ferrari, Pierangela Samarati, Elisa Bertino, and Sushil Jajodia. Providing flexibility in information flow control for object-oriented systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 130–140, Oakland, CA, USA, May 1997.

[HR98] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, San Diego, California, January 1998.

[JD96] Daniel Jackson and Craig A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7):484–495, July 1996.

[JL78] Anita K. Jones and Barbara Liskov. A language extension for expressing constraints on data access. *Comm. of the ACM*, 21(5):358–367, May 1978.

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, Saint-Malo, France, 1997.

[MMN90] Catherine J. McCollum, Judith R. Messing, and LouAnna Notargiacomo. Beyond the pale of MAC and DAC — defining new forms of access control. In *Proc. IEEE Symposium on Security and Privacy*, pages 190–200, 1990.

[MR92] M. D. McIlroy and J. A. Reeds. Multilevel security in the UNIX tradition. *Software—Practice and Experience*, 22(8):673–694, August 1992.

[PO95] Jens Palsberg and Peter Ørbæk. Trust in the λ-calculus. In *Proc. 2nd International Symposium on Static Analysis*, number 983 in Lecture Notes in Computer Science, pages 314–329. Springer, September 1995.

[RSC92] Joel Richardson, Peter Schwarz, and Luis-Felipe Cabrera. CACL: Efficient fine-grained protection for objects. In *Proceedings of the 1992 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 154–165, Vancouver, BC, Canada, October 1992.

[Sal74] J. H. Saltzer. Protection and the control of information sharing in Multics. *Comm. of the ACM*, 17(7):388–402, July 1974.

[SV98] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, San Diego, California, January 1998.

[Vol97] Dennis Volpano. Provably-secure programming languages for remote evaluation. *ACM SIGPLAN Notices*, 32(1):117–119, January 1997.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.