Protecting Privacy using the Decentralized Label Model

Andrew C. Myers
Cornell University
and
Barbara Liskov
Massachusetts Institute of Technology.

Stronger protection is needed for the confidentiality and integrity of data, because programs containing untrusted code are the rule rather than the exception. Information flow control allows the enforcement of end-to-end security policies but has been difficult to put into practice. This paper describes the decentralized label model, a new label model for control of information flow in systems with mutual distrust and decentralized authority. The model improves on existing multilevel security models by allowing users to declassify information in a decentralized way, and by improving support for fine-grained data sharing. It supports static program analysis of information flow, so that programs can be certified to permit only acceptable information flows, while largely avoiding the overhead of run-time checking. The paper introduces the language Jif, an extension to Java that provides static checking of information flow using the decentralized label model.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—*information flow controls*

General Terms: Security, Information flow controls, Programming languages

Additional Key Words and Phrases: confidentiality, declassification, end-to-end, downgrading, integrity, lattice, policies, principals, roles, type checking

This research was supported by DARPA Contracts F30602-96-C-0303, F30602-98-1-0237, and F30602-99-1-0533, monitored by USAF Rome Laboratory. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.
Name: Andrew C. Myers
Affiliation: Dept. of Computer Science, Cornell University
Address: (607) 255-8597, andru@cs.cornell.edu, 4130 Upson Hall, Ithaca NY 14853
Name: Barbara Liskov
Affiliation: Laboratory for Computer Science, Massachusetts Institute of Technology
Address: (617) 253-5886, liskov@lcs.mit.edu, 545 Technology Sq., Cambridge, MA 02139

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept, ACM Inc., 1515 Broadway, New York, NY 10036 USA, fax +1 (212) 869-0481, or `permissions@acm.org`.

1. INTRODUCTION

The common models for computer security are proving inadequate. Security models have two goals: preventing accidental or malicious destruction of information, and controlling the release and propagation of that information. Only the first of these goals is supported well at present, by security models based on access control lists or capabilities (i.e., *discretionary access control*, simply called "access control" from this point on). Access control mechanisms do not support the second goal well: they help to prevent information release but do not control information propagation. If a program is allowed to read some confidential data, it is difficult to control how it distributes the data it has read without impractically restricting its communication. What is needed are *end-to-end* confidentiality policies, which differ from the policies enforced by conventional access control mechanisms in that the confidentiality of data is protected even if the data is manipulated and transformed by untrusted programs.

Systems that support the downloading of untrusted code are particularly in need of a better security model. For example, Java [Gosling et al. 1996] supports downloading of code from remote sites, which creates the possibility that the downloaded code will transfer confidential data to those sites. Java attempts to prevent these transfers by using its compartmental *sandbox* security model, but this approach largely prevents applications from sharing data. Different data manipulated by an application have different security requirements, but a compartmental model restricts all data equally.

End-to-end security policies require more precise control of information propagation, which is supported by existing mandatory access control models [Bell and LaPadula 1975; Department of Defense 1985]; however, these models have high run-time overhead and unduly restrict what computations can be performed. Many models for more precise characterization of information flow have been developed (*e.g.*, [Sutherland 1986; McLean 1990; Wittbold and Johnson 1990; Gray 1991; McLean 1994]) but are difficult to put into practice. The goal of this work is to make information flow control more practical than was previously possible.

The *decentralized label model* addresses the weaknesses of earlier approaches to the protection of confidentiality in a system containing untrusted code or users, even in situations of mutual distrust. The model allows users to control the flow of their information without imposing the rigid constraints of a traditional multilevel security system. It protects confidentiality for users and groups rather than for a monolithic organization.[1] The decentralized label model also introduces a richer notion of *declassification*. All practical information flow control systems provide the ability to arbitrarily weaken information flow restrictions, because strict information flow control is too restrictive for writing real applications. For many computations, some amount of information leakage is both necessary and acceptable; information-theory techniques can be used in some cases to rigorously bound the amount of information leaked [Millen 1987; Karger and Wray 1991; Kang et al. 1995].

However, declassification in earlier systems for controlling information flow lies outside the model: it is performed by a *trusted subject*, which possesses the authority of a universally trusted principal. Traditional information flow control is thus not workable in a decentralized environment; the notion of a universally trusted principal does not make

[1] In this paper, the terms *privacy*, *confidentiality*, and *secrecy* will be considered synonyms, although "privacy" is also used in other work in a more general sense or as a synonym for *anonymity*.

sense there, nor does the idea that a data item has a single universally accepted security classification. Declassification in the decentralized label model has the novel feature that it permits principals to declassify their own data, rather than requiring a trusted subject to do it. Information flow in a decentralized system is controlled in accordance with policies from potentially many sources; in this model, a principal may weaken only the policies that it has itself provided, and thus may not endanger data that it does not own.

To control information flow with fine granularity, it must be tracked at the programming language level, and as we will see later, static analysis of programs is needed to control some information flows adequately. The decentralized label model has been applied to a new programming language called Jif, for Java Information Flow [Myers 1999a][2].

The decentralized label model defines a set of rules that programs must follow in order to avoid leaks of private information. In Jif, these rules can largely be checked statically. User-supplied program annotations (labels) describe the allowed flow of information in a Jif program. These labels, in conjunction with ordinary Java type declarations, form an extended type system. Jif programs are then type-checked at compile time, to ensure not only that they are type-safe, but also that they do not violate information flow rules. Compile-time checks of course impose no run-time overhead in space or time, but they have the additional advantage over run-time checks that when they fail, they do not leak information about the data the program is using. Because information flow restrictions are enforced by type checking, this enforcement mechanism also enjoys the same compositional property that ordinary type checking does: two independently checked modules, joined together, can be treated as a single secure module as long as each module uses the other in accordance with the (extended) type system.

An important philosophical difference between Jif and other work on static checking of information flow is the focus on a usable programming model. One problem with previous models of static flow analysis is that they are too limited or too restrictive to be used in practice. The goal of Jif is to add enough power to the static checking framework to allow reasonable programs to be written in a natural manner. This model of secure computation should be widely applicable, both to provide assurance for systems that protect the private information of their clients against mistaken leaks of their information, and for protection in systems that run untrusted code such as applets and servlets.

Jif does have some limitations. Its static flow analysis is intended to control covert and legitimate storage channels; it does not control certain covert channels that are more difficult to identify at the level of the programming language. However, some covert channels (particularly timing channels) are ruled out in Jif by the single-threaded structure of the language.

Another assumption of Jif is that programs must run on a *trusted execution platform* that enforces the rules of the model. The number of components in the trusted execution platform should be as small as possible. In the current implementation of this model, the Jif compiler statically checks information flows in a program before it is allowed to execute. Although techniques for proving compilers correct are improving, we would prefer not to have to trust the compiler. One fix that requires less trust is to use a verifier that checks the output of such a compiler; Section 8 discusses how this might be done.

With conventional security mechanisms, all programs are part of the trusted computing base with respect to the protection of confidentiality, since there is no internal mechanism

[2]This language was earlier known as JFlow

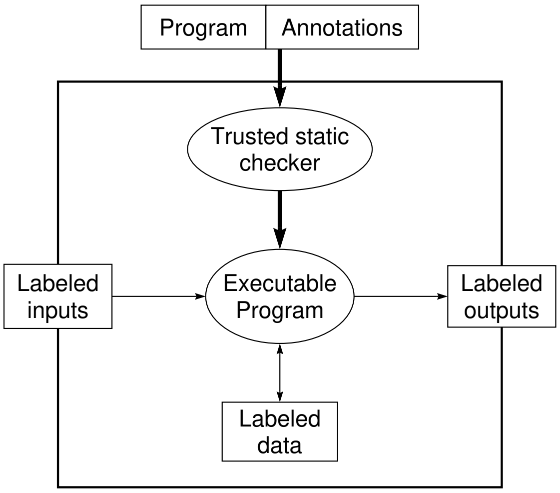

Fig. 1. Trusted execution platform

other than discretionary access control ensuring that programs respect confidentiality. In Jif, the model is that the static checker (or verifier) rejects programs containing information flows that might violate confidentiality requirements.

The trusted computing base also must include many of the usual trusted components of an operating system: hardware that has not been subverted, a trustworthy underlying operating system, and a reliable authentication mechanism. Together, these trusted components make a trusted execution platform. Figure 1 depicts a trusted execution platform, into which code may enter (along the heavy arrows) only if it has been checked statically by a trusted static checker to ensure that it obeys the label model. Data manipulated by the executable program is labeled, as are inputs to and outputs from the system.

The remainder of the paper describes the model and briefly describes the Jif language with an example. More details on the model, the Jif language, and the static checking of Jif code is available elsewhere [Myers and Liskov 1998; Myers 1999a; Myers 1999b].

The organization of the paper is as follows. Section 2 briefly describes some systems that can benefit from decentralized information flow control, and which are not well supported by existing models. Section 3 introduces the fundamentals of the new information flow control model. Section 4 extends the decentralized label model to express constraints on data integrity as well as confidentiality. Section 5 shows how the model can be integrated into a expressive programming language. Section 6 describes related work in the areas of information flow models, access control, and static program analysis. Section 7 concludes and future work is discussed in Section 8.

2. CONFIDENTIALITY EXAMPLE

Figure 2 depicts an example with security requirements that cannot be satisfied using existing techniques. This scenario contains mutually distrusting principals that must cooperate to perform useful work. In the example, the user Bob is preparing his tax form using both a spreadsheet program and a piece of software called "WebTax". Bob would like to be able to prepare his final tax form using WebTax, but he does not trust WebTax to respect his privacy. The computation is being performed using two programs: a spreadsheet that

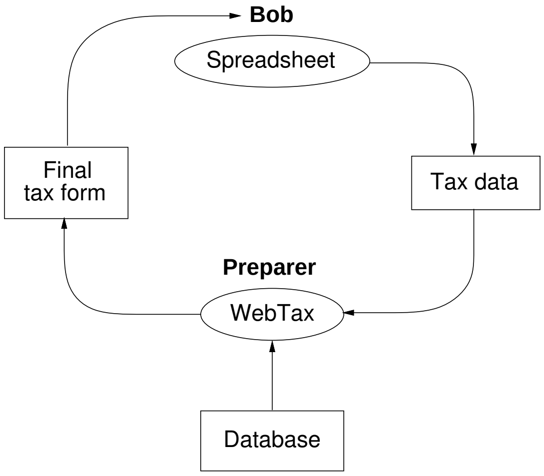

Fig. 2. A simple example

Bob trusts and to which he grants his full authority, and the WebTax program that Bob does not trust. Bob would like to transmit his tax data from the spreadsheet to WebTax and receive a final tax form as a result, while being protected against WebTax leaking his tax information.

In this example, there is another principal named Preparer that also has privacy interests. This principal represents a firm that distributes the WebTax software. The WebTax application computes the final tax form using a proprietary database, shown at the bottom, that is owned by Preparer. This database might, for example, contain secret algorithms for minimizing tax payments. Since this principal is the source of the WebTax software, it trusts the program not to distribute the proprietary database through malicious action, though the program might leak information because it contains bugs.

It may be difficult to prevent some information about the database contents from leaking back to Bob, particularly if Bob is able to make a large number of requests and then carefully analyze the resulting tax forms. This information leak is not a practical problem if Preparer can charge Bob a per-form fee that exceeds the value of the information Bob obtains through each form.

To make this scenario work, Preparer needs two pieces of functionality. First, it needs protection against accidental or malicious release of information from the database by paths other than through the final tax form. Second, it needs the ability to *sign off* on the final tax form, confirming that the information leaked in the final tax form is sufficiently small or scrambled by computation that the tax form may be released to Bob.

It is worth noting that Bob and Preparer do need to trust that the execution platform has not been subverted. For example, if WebTax is running on a computer that Bob completely controls, then Bob will be able to steal the proprietary database. Clearly, Preparer cannot have any real expectation of confidentiality if its private data is manipulated in unencrypted form by an execution platform that it does not trust!

Even if the execution platform is trusted, this scenario cannot be implemented satisfactorily using existing security techniques. With the currently standard security techniques, Bob must carefully inspect the Webtax code and verify that it does not leak his data; this is difficult. The techniques described in this paper allow the security goals of both Bob and Preparer to be met without this manual inspection; Bob and Preparer can then cooperate in performing useful computation. In another sense, this work shows how both Bob and Preparer can inspect the Webtax program efficiently and simply to determine whether it violates their security requirements.

3. MODEL

This section describes the decentralized label model. The key new feature of this model is that it supports computation in an environment with mutual distrust. In a decentralized environment, security policies cannot be decided by any central authority. Instead, individual participants in the system must be able to define and control their own security policies. The system will then enforce behavior that is in accordance with all of the security policies that have been defined. One innovation that makes this approach possible is the addition of a notion of ownership to the labels used to annotate data. These labels make new functionality possible: a principal may chose to selectively declassify its own data, but does not have the ability to weaken the policies of other principals.

The decentralized label model also provides a more expressive and flexible language for describing allowed information flows than earlier models. It allows information flow policies to be defined conveniently in terms of principals representing groups and roles. The rule for relabeling data also has been shown to be both sound and complete with respect to a simple formal semantics for labels: the rule only allows relabelings that are safe in this semantics, and it allows *all* safe relabelings [Myers and Liskov 1998].

The essentials of the decentralized label model are principals, which are the entities whose privacy is protected by the model; and *labels*, which are the way that principals express their privacy concerns. In addition, there are rules that must be followed as computation proceeds in order to avoid information leaks, including the selective declassification mechanism.

In this model, users are assumed to be external to the system on which programs run. Information is leaked only when it leaves the system. The mere manipulation of data by a program—regardless of the privileges possessed by that program—is not considered to be a release of information to any principal. The model also contains rules governing the release of information to the external environment, as well as rules for reading information from the environment.

3.1 Principals

In the decentralized label model, information is owned by, updated by, and released to *principals*: users and other authority entities such as groups or roles. For example, both users and groups in Unix would be represented by principals. A process has the authority to act on behalf of some set of principals.

Some principals are authorized to *act for* other principals. When one principal p can act for another principal p', p possesses all the potential powers and privileges of p'. The *acts for* relation is reflexive and transitive, defining a hierarchy or partial order of principals. For compactness, the statement "p acts for q" is written formally as $p \succeq q$. This relation is similar to the *speaks for* relation [Lampson et al. 1991]; the principal hierarchy also is similar to a *role hierarchy* [Sandhu 1996]. Although the principal hierarchy changes over time, revocation of acts-for relations is assumed to occur infrequently.

The acts-for relation can be used to model groups and roles conveniently, as shown

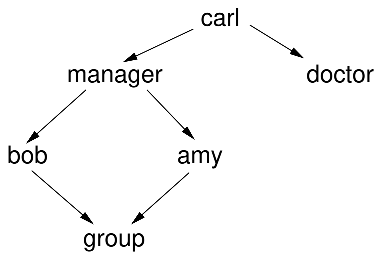

Fig. 3. A principal hierarchy

in Figure 3. In this example, a group named group is modeled by introducing acts-for relationships between each of the group members (amy and bob) and the group principal. These relationships allows Amy and Bob to read data readable by the group and to control data controlled by the group. A principal manager representing Bob and Amy's manager is able to act for both amy and bob. This principal is one of the roles that a third user, Carl (carl), is currently enabled to fulfill; Carl also has a separate role as a doctor. Carl can use his roles to prevent accidental leakage of information between data stores associated with his different jobs.

The acts-for relation permits delegation of all of a principal's powers, or none. However, the principle of least privilege suggests that it is desirable to separate out the various powers that may be given by one principal to another. For example, the acts-for relation can be separated into *declassifies-for* and *reads-for* relations between principals, as well as other relations less directly connected to information flow control. However, the model presented here can be extended to support some finer-grained relations on principals in a natural manner, by representing the finer-grained relations as simple acts-for relations on a more complex principal hierarchy [Myers 1999b].

The principal hierarchy defines an important part of the complete security policy for the system. Importantly, it can also be managed in a decentralized fashion. The relationship $p \succeq q$ can be added to the principal hierarchy as long as the adding process has sufficient privilege to act for the principal q in this manner. No privileges are needed with respect to p because this relationship only empowers p. Thus, there is no intrinsic need to centralize the mechanism that controls changes to the principal hierarchy.

3.2 Labels

Principals express their privacy concerns by using *labels* to annotate programs and data. A label is a set of components that express privacy requirements stated by various principals. A component has two parts, an owner and a set of readers, and is written in the form *owner*: *readers*. The purpose of a label component is to protect the privacy of the owner of the component. The readers of a component are the principals that this component permits to read the data. Thus, the owner is a source of data, and the readers are possible destinations for the data. Principals not listed as readers are not permitted to read the data.

Because each component of a label controls the uses of data with that label, a label component is called a *policy* for use of the data. Thus, a data item is labeled with some number of policies stated by the owning principals. The use of the term "policy" is intended to be suggestive but should not be confused with its use for the high-level specification of information flows allowed within the system as a whole (*e.g.*, *non-interference policies* [Goguen and Meseguer 1984]). The policies in decentralized labels, by contrast, apply only to a single data value.

An example of a label is $L = \{o_1 : r_1, r_2;\ \ o_2 : r_2, r_3\}$. Here, o_1, o_2, r_1, r_2 denote principals. Semicolons separate two policies (components) within the label L. The owners of these policies are o_1 and o_2, and the reader sets of the policies are $\{r_1, r_2\}$ and $\{r_2, r_3\}$, respectively.

The intuitive meaning of a label is that every policy in the label must be obeyed as data flows through the system, so that labeled information is released only by the consensus of all of the policies. A user may read the data only if the user's principal can act for a reader for *each* policy in the label. Thus only users whose principals can act for r_2 can read the data labeled by L.

The same principal may be the owner of several policies; the policies are still enforced independently of one another in this case. If a principal is the owner of *no* policies in the label, it is as if the label contained a policy with that owner, but every possible principal listed as a reader.

This label structure allows each owner to specify an independent flow policy, and thus to retain control over the dissemination of its data. Code running with the authority of an owner can modify a policy with that owner; in particular, the program can *declassify* that data by adding additional readers. Since declassification applies on a per-owner basis, no centralized declassification process is needed, as it is in systems that lack ownership labeling.

Labels are used to control information flow within programs by annotating program variables with labels. The label of a variable controls how the data stored within that variable can be disseminated. The key to protecting confidentiality is to ensure that as data flows through the system during computation, its labels only become more restrictive: the labels have more owners, or particular owners allow fewer readers. If the contents of one variable affect the contents of another variable, there is an information flow from the first variable to the second, and therefore, the label of the second variable must be at least as restrictive as the label of the first.

This condition can be enforced at compile time by type checking, as long as the label of each variable is known at compile time. In other words, the label of a variable cannot change at run time, and the label of the data contained within a variable is always the same as the label of the variable itself. If the label of a variable could change, the label would need to be stored and checked at run time, creating an additional information channel and leading to significant performance overhead. Immutable variable labels might seem to be a limitation, but mechanisms discussed later restore expressiveness.

When a value is read from a variable x, the apparent label of the value is the label of x; whatever label that value had at the time it was written to x is no longer known when it is read. In other words, writing a value to a variable is a relabeling, and is allowed only when the label of the variable is at least as restrictive as the apparent label of the value being written.

Giving private data to an untrusted program does not create an information leak—even if that program runs with the authority of another principal—as long as that program obeys all of the label rules described here. Information can be leaked only when it leaves the system through an *output channel*, so output channels are labeled to prevent leaks. Information can enter the system through an *input channel*, which also is labeled to prevent leaks of data that enters. It is not necessarily an information leak for a process to manipulate data even though no principal in its authority has the right to read it, because all the process can do is write the data to a variable or a channel with a label that is at least as restrictive as the data's label.

3.3 Incremental relabelings

A value may be assigned to a variable only if the relabeling that occurs at this point is a *restriction*, a relabeling in which the set of readers permitted by the new label is in all contexts a subset of the readers permitted by the original. This condition is satisfied, for example, if the two labels differ only in a single policy, and the readers of that policy in the label of the variable are a subset of the readers of that policy in the label of the value. For example, a relabeling from $\{\mathsf{bob} : \mathsf{amy}, \mathsf{carl}\}$ to $\{\mathsf{bob} : \mathsf{amy}\}$ is a restriction because the set of readers allowed by bob becomes smaller. If a relabeling is a restriction, it is considered to be *safe*.

Clearly, a change to a label in which a reader r is removed cannot make the changed policy any less restrictive, and therefore this change is safe. Note that the removal of a reader does not necessarily make the policy *more* restrictive if there is another reader in the policy for which r acts. There are four different ways that a label can be safely changed:

—**Remove a reader.** It is safe to remove a reader from some policy in the label, as just described.

—**Add a policy.** It is safe to add a new policy to a label; since all previous policies are still enforced, the label cannot become less restrictive.

—**Add a reader.** It is safe to add a reader r' to a policy if the policy already allows a reader r that r' acts for. This change is safe because if r' acts for r, it is already effectively considered to be a reader: r' has all of the privileges of r anyway.

—**Replace an owner.** It is safe to replace a policy owner o with some principal o' that acts for o. This change is safe because the new policy allows only processes that act for o' to weaken it through declassification, while the original policy also allows processes with the weaker authority of o to declassify it.

Under an alternate semantics for labels in which the owner of a policy is implicitly allowed as a reader, it is safe to add the owner of a policy as an explicit reader of its own policy. This incremental relabeling step allows certain common labels to be expressed more concisely, although it has no interesting effects on the properties of labels [Myers 1999b].

3.4 The relabeling rule

These incremental relabelings capture all of the safe modifications that can be made to a label. A relabeling from a label L_1 to a label L_2 will be safe if there is some sequence of these incremental relabelings that rewrites L_1 into L_2. To allow efficient label checking of programs, a rule is needed to determine whether such a sequence exists, given two labels. This section presents such a rule.

We can define precisely when a relabeling from label L_1 to label L_2 is a restriction with the aid of some additional notation. If the relabeling is a restriction, then either L_1 is as restrictive as L_2, or possibly less restrictive. In other words, L_1 is at most as restrictive as L_2, and conversely L_2 is at least as restrictive as L_1. The notation $L_1 \sqsubseteq L_2$ is used to denote this relationship.

1. $\{\mathsf{amy} : \mathsf{bob}, \mathsf{carl}\} \sqsubseteq \{\mathsf{amy} : \mathsf{carl}\}$
2. $\{\mathsf{amy} : \mathsf{bob}\} \sqsubseteq \{\mathsf{amy} :\ \}$
3. $\{\mathsf{amy} : \mathsf{manager}\} \sqsubseteq \{\mathsf{amy} : \mathsf{carl}\}$
4. $\{\mathsf{manager} : \mathsf{bob}\} \sqsubseteq \{\mathsf{carl} : \mathsf{bob}\}$
5. $\{\mathsf{amy} : \mathsf{carl}\} \not\sqsubseteq \{\mathsf{amy} : \mathsf{bob}\}$
6. $\{\mathsf{amy} : \mathsf{carl}\} \not\sqsubseteq \{\mathsf{bob} : \mathsf{carl}\}$
7. $\{\mathsf{amy} : \mathsf{manager}\} \not\sqsubseteq \{\mathsf{amy} : \mathsf{bob}\}$
8. $\{\mathsf{manager} : \mathsf{bob}\} \not\sqsubseteq \{\mathsf{bob} : \mathsf{bob}\}$

Fig. 4. Examples of policy relationships

It is clear that L_2 is at least as restrictive as L_1 if every policy I in L_1 is guaranteed to be enforced by L_2. This guarantee turns out to hold exactly when there is some single policy J in L_2 that is guaranteed to enforce the policy I. For brevity, we will say that J *covers* I. For example, the label $L_1 = \{\mathsf{bob} : \mathsf{amy}, \mathsf{bob}\}$ can be relabeled to the label $L_2 = \{\mathsf{bob} : \mathsf{bob}; \mathsf{amy} : \mathsf{carl}\}$ because the single policy $(\mathsf{bob} : \mathsf{amy}, \mathsf{bob})$ in L_1 is guaranteed to be enforced by the first policy of L_2, which has the same owner and fewer readers. The label L_2 contains an additional policy $(\mathsf{amy} : \mathsf{carl})$, but because information can flow only in accordance with every policy in a label, the addition of this second policy only makes L_2 more restrictive still.

In general, it may be the case that a label contains two policies such that one policy covers the other. In this case, the covered policy is redundant and the label is equivalent to one in which the covered policy is removed. This simplification is important for the performance of label checking, although it never affects correctness.

The rule for when one label is more restrictive than another is built upon the notion of one policy covering another. It is important for expressive power that the rule for policies be as permissive as possible while remaining safe. The rule with this property is that a policy I is covered by J (which is written $I \sqsubseteq J$) if two conditions hold. The first condition is that the owner of the policy J must either be the same as the owner of the policy I, or be able to act for the owner of I. The intuition behind this condition is that the owner of the policy is the least powerful principal able to declassify the policy. For the new policy J to be no less restrictive, this principal must either remain the same or increase in power. The second condition is that the readers allowed by J must be a subset of the readers allowed by I, where we consider that the readers allowed by a policy include not only the principals explicitly mentioned by the policy but also any principal able to act for the explicitly mentioned readers. The intuition here is that a principal that can act for an explicitly mentioned reader is able to read the data.

In Figure 4, some examples of this rule are shown, using the principal hierarchy of Figure 3. These examples show cases when one label is (1–4) or is not (5–8) at most as restrictive as another, for labels containing only a single policy each. When two labels each contain only a single policy, the rule for restrictiveness on labels reduces to the rule for restrictiveness on the respective policies, so these examples illustrate the policy restriction rule. In Examples 1 and 2, the reader bob has been removed from the reader set, which is safe. In Example 3, the reader carl has been added and the reader $\mathsf{manager}$ then removed. In Example 4, the owner $\mathsf{manager}$ has been replaced by the owner carl.

With the aid of some additional notation, these relationships among labels and policies can be defined formally. The notation $\mathbf{o}(I)$ denotes the owner of a policy I, and the notation $\mathbf{r}(I)$ denotes the set of readers of I. If a principal p_1 acts for a principal p_2 in the principal hierarchy in which the relabeling is being checked, we will write $p_1 \succeq p_2$. Given a policy I, we can define a function $\mathbf{R}$ that yields the set of principals implicitly allowed as readers by that policy:

$$\mathbf{R}(I) = \{p \mid \exists_{p' \in \mathbf{r}(I)}\; p \succeq p'\}$$

This function can be used to define when one label is at most as restrictive as another ($L_1 \sqsubseteq L_2$) and when one policy is at most as restrictive as (that is, covers) another ($I \sqsubseteq J$):

Definition of the complete relabeling rule ($\sqsubseteq$)

$$\begin{aligned} L_1 \sqsubseteq L_2 &\equiv \forall_{I \in L_1}\; \exists_{J \in L_2}\; I \sqsubseteq J \\ I \sqsubseteq J &\equiv \mathbf{o}(J) \succeq \mathbf{o}(I) \;\wedge\; \mathbf{R}(J) \subseteq \mathbf{R}(I) \\ &\equiv \mathbf{o}(J) \succeq \mathbf{o}(I) \;\wedge\; \forall_{p' \in \mathbf{r}(J)}\; \exists_{p \in \mathbf{r}(I)}\; p' \succeq p \end{aligned}$$

This rule is called the *complete relabeling rule* because it defines exactly when relabeling from L_1 to L_2 is safe. The difficulty in showing the safety of the relabeling rule is that relabeling is checked statically (perhaps at compile time), and the actual run-time principal hierarchy may differ from that observed at the time of the check (the *static principal hierarchy*).

This proof that the relabeling rule is complete, and a corresponding proof that it is sound, have been presented elsewhere with respect to a simple formal semantics for labels [Myers and Liskov 1998; Myers 1999b]. Because the relabeling rule is sound, a relabeling checked at compile time is guaranteed to be a restriction even in the run-time principal hierarchy. Because the relabeling rule is complete, it is the most permissive sound relabeling rule. In other words, the relabeling rule is as expressive as possible while remaining safe; this expressiveness is useful in modeling many information flow scenarios.

3.5 Computing values

During computation, values are derived from other values. Since a derived value may contain information about its sources, its label must enforce the policies of each of its sources. For example, if we multiply two integers, the product's label must be at least as restrictive as the labels of both operands. However, to avoid unnecessary restrictiveness, the label of the product should be the *least* restrictive label having this property.

When a program combines two values labeled with L_1 and L_2, respectively, the result should have the least restrictive label that enforces all the information flow restrictions specified by L_1 and L_2. Recall that a label is simply a set of policies. The least restrictive set of policies that enforces all the policies in L_1 and L_2 is simply the union of the two set of policies. This least restrictive label is the *least upper bound* or *join* of L_1 and L_2, written as $L_1 \sqcup L_2$ (The symbol $\oplus$ also has been used to denote the join of two security classes [Denning and Denning 1977; Andrews and Reitman 1980]). Note that $L_1 \sqsubseteq L_1 \sqcup L_2$ for all labels L_1 and L_2: Joining is always a restriction of both input labels. The rule for the join of two labels can be written very compactly by considering L_1 and L_2 as sets of policies:

Labels for Derived Values (Definition of $L_1 \sqcup L_2$)

$$L_1 \sqcup L_2 = L_1 \cup L_2$$

It may be the case that some policies in the union of the two sets are redundant; in this case, they can be removed. For example, the join of the labels {amy : bob} and

$\{\mathsf{amy} : \mathsf{bob}, \mathsf{carl}\}$ is $\{\mathsf{amy} : \mathsf{bob};\ \mathsf{amy} : \mathsf{bob}, \mathsf{carl}\}$, which is equivalent to the label $\{\mathsf{amy} : \mathsf{bob}\}$, because the second policy in the union is covered by the first, regardless of the principal hierarchy.

3.6 Declassification

Because labels in this model contain information about the owners of labeled data, these owners can retain control over the dissemination of their data, and relax overly restrictive policies when appropriate. This second kind of relabeling is a selective form of *declassification*.

The ability of a process to declassify data depends on the authority possessed by the process. At any moment while executing a program, a process is authorized to act on behalf of some (possibly empty) set of principals. This set of principals is referred to as the *authority* of the process. If a process has the authority to act for a principal, actions performed by the process are assumed to be authorized by that principal. Code running with the authority of a principal can declassify data by creating a copy in whose label a policy owned by that principal is relaxed. In the label of the copy, readers may be added to the reader set, or the policy may be removed entirely, which is effectively the same as adding all principals as readers in the policy.

Because declassification applies on a per-owner basis, no centralized declassification process is needed, as it is in systems that lack ownership labeling. Declassification is limited because it cannot affect the policies of owners the process does not act for; declassification is safe for these other owners because reading occurs only by the consensus of all owners.

The declassification mechanism makes it clear why the labels maintain independent reader sets for each owning principal. If, instead, a label consisted of just an owner set and a reader set, information about the individual flow policies would be lost, reducing the power of declassification.

Because the ability to declassify depends on the run-time authority of the process, it requires a run-time check for the proper authority. Declassification seems to be needed infrequently, so the overhead of this run-time check is not large. It can be reduced still further using the proper static framework [Myers 1999a].

Declassification can be described more formally. A process may weaken or remove any policies owned by principals that are part of its authority. Therefore, the label L_1 may be relabeled to L_2 as long as $L_1 \sqsubseteq L_2 \sqcup L_A$, where L_A is a label containing exactly the policies of the form $\{p :\}$ for every principal p in the current authority. The rule for declassification may be expressed as an inference rule:

Relabeling by declassification

$$\frac{L_A = \bigsqcup_{(p \text{ in current authority})} \{p :\ \} \qquad L_1 \sqsubseteq L_2 \sqcup L_A}{L_1 \text{ may be declassified to } L_2}$$

This inference rule builds on the rule for relabeling by restriction. The subset rule for relabeling L_1 to L_2 states that for all policies J in L_1, there must be a policy K in L_2 that is at least as restrictive. The declassification rule has the intended effect because for policies J in L_1 that are owned by a principal p in the current authority, a more restrictive policy K is found in L_A. For other policies J, the corresponding policy K must be found in L_2, since the current authority does not have the power to weaken them. This rule also shows that a label L_1 always may be declassified to a label that it could be relabeled to by restriction, because the relabeling condition $L_1 \sqsubseteq L_2$ implies the declassification condition $L_1 \sqsubseteq L_2 \sqcup L_A$.

We have now seen three fundamental rules for the manipulation of labels during the execution of a program: rules for relabeling, derived labels, and selective declassification. Analysis of the information flow in a piece of code reduces to analysis of the uses of these rules. As we will see in Section 5, the rules can be checked at compile time, assuming the programmer is willing to supply some annotations. Code that performs a relabeling that is a restriction can be checked entirely at compile time, but code that performs a declassification requires a run-time test that the process acts for the owner.

3.7 Channels

Input and output channels allow data to enter and leave the domain in which the label rules are enforced. Channels are half-variables; like variables, they have an associated label and can be used as an information conduit. However, they provide only half the functionality that a variable provides: either input or output. As with a variable, when a value is read from an input channel, the value acquires the label of the input channel. Similarly, a value may be written to an output channel only if the label of the output channel is at least as restrictive as the label on the value; otherwise, an information leak is presumed to occur.

Obviously, the assignment of labels to channels is a security-critical operation, and it is important that the label of a channel reflects reality. For example, if the output of a printer is known to be readable by a number of people, confidentiality may be violated unless the label of the output channel to that printer identifies all of them, perhaps by using a group principal. If two computers separately enforcing the decentralized label model communicate over channels, it is important to choose the labels of the corresponding output and input channels carefully so that labels cannot be laundered by a round trip.

Labels of input and output channels to devices like displays and printers typically have a single owner (*e.g.*, root) that allows as readers all principals representing actual users of the device. Labels of channels representing communication over a network, however, may have more than one owner.

Because the output channel has a decentralized label, there does not need to be any notion of the output channel's destination that is accepted by all principals in the system, because the effect of the relabeling rules is that a principal p accepts the reader set of a policy only if the owner of the policy acts for p.

In fact, the process of creating labeled output channels requires only one additional mechanism: the ability to create a *raw output channel*, which is an output channel with the label $\{\}$. Data can be written to such a channel only if it has no privacy restrictions, so the creation of such a channel cannot in itself result in the leaking of any private data. A output channel labeled L is constructed from a raw output channel by writing a procedure that declassifies data labeled with L to the label $\{\}$. This procedure can be added to the system only by a principal with the ability to act for all the owners in L. In other words, the validation of output channels does not need to be performed by a trusted component of the system. In practice, it likely will be useful to have a widely accepted validation process.

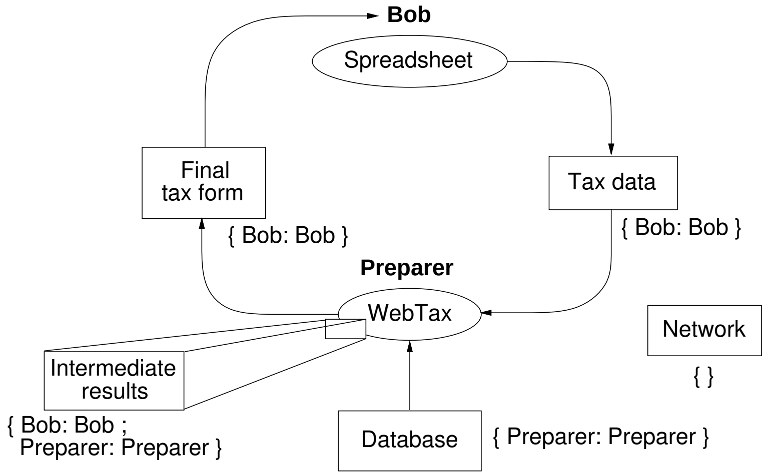

Fig. 5. Annotated Tax Preparation Example

3.8 Confidentiality example revisited

The tax preparer example from Section 2 is repeated in Figure 5, except that all data in the example has been annotated with labels to protect the privacy of the principals Bob and Preparer. It can be seen that these labels obey the rules given and meet the security goals set out for this scenario.

In the figure, ovals indicate programs executing in the system. A boldface label beside an oval indicates the authority with which a program acts. In this example, the principals involved are Bob and Preparer, as we have already seen, and they give their authority to the spreadsheet and WebTax programs, respectively. Arrows in the diagrams represent information flows between principals; square boxes represent the data being transmitted.

First, Bob applies the label {Bob: Bob} to his tax data. This label allows no one to read the data except Bob himself. With this label applied to it, tax data cannot be sent to an untrusted network location (represented as an output channel with the label {}) because it is not the case that {Bob: Bob} $\sqsubseteq$ {}. Bob can give this data to the WebTax program with confidence that it cannot be leaked, because WebTax will be unable to remove the {Bob: Bob} policy from the tax data or any data derived from it.

The WebTax program uses Bob's tax data and its private database to compute the tax form. Any intermediate results computed from these data sources will have the label {Bob: Bob; Preparer: Preparer}. Because the reader sets of this label disagree, the label prevents both Bob and Preparer (and everyone else) from reading the intermediate results. This joint label is generated by the rule for join:

$$\{\mathsf{Bob} : \mathsf{Bob}\} \sqcup \{\mathsf{Preparer} : \mathsf{Preparer}\} =$$

$$\{\mathsf{Bob} : \mathsf{Bob}\ ;\ \mathsf{Preparer} : \mathsf{Preparer}\}$$

This label protects Preparer against accidental disclosure of its private database through programming errors in the WebTax application.

Before being released to Bob, the final tax form has the same label as the intermediate results, and is not readable by Bob, appropriately. To make the tax form readable, the

WebTax application *declassifies* the label by removing the {Preparer: Preparer} policy. The application can do this because the Preparer principal has granted the application its authority. This grant of authority is reasonable because Preparer supplied the application and presumably trusts that it will not use the power maliciously.

The authority to act as Preparer need not be possessed by the entire WebTax application, but only by the part that performs the final release of the tax form. By limiting this authority to a small portion of the application, the risk of accidental release of the database is reduced. Thus the WebTax application might have a small top-level routine that runs with the authority of Preparer, while the rest of its code runs with no authority.

3.9 Lattice properties

The set of labels forms a pre-order with the essential properties of a security-class lattice [Denning 1976]. Each element in the lattice is one of the possible labels. Labels exist in a partial order as defined by the restriction relation, $\sqsubseteq$. The least restrictive label, written as $\{\}$, corresponds to data that can flow anywhere; the greatest possible restriction, $\top$, corresponds to data that can flow nowhere: it is readable by no one and owned by everyone. As we go up in the lattice, labels become strictly more restrictive. Data always can be relabeled upward in the lattice, because restriction does not create a possible information leak.

An important property of these rules is that the join operator does not interfere with relabeling or declassification. It is possible to relabel a component of a join independently: if the relabeling $L_1 \rightarrow L_2$ is legal, then for any other label L_3, the relabeling $L_1 \sqcup L_3 \rightarrow L_2 \sqcup L_3$ is also legal. This property automatically holds for ordinary relabeling because of the lattice properties of the set of labels. The property also holds for declassification because the join operator ensures that the policies in the label L_3 are not violated.

The set of labels is not precisely a lattice because it is possible for two unequal labels L_1 and L_2 to be equivalent, in the sense that $L_1 \sqsubseteq L_2$ and $L_2 \sqsubseteq L_1$. For example, simplification of a label by removal of a redundant policy results in a different, but equivalent label. The *anti-symmetry* property of lattices requires equality of equivalent elements; however, labels possess all the lattice properties that are necessary for reasoning about information flow.

Declassification introduces an exception to the simple lattice model of security classes, because each principal has access to certain relabelings that do not agree with the lattice. However, these relabelings do not leak information owned by other principals in the system. Considering declassification as a relabeling, each principal effectively sees a somewhat different lattice of labels.

4. INTEGRITY LABELS

We have seen that the decentralized label model supports labels containing privacy policies. This model can be generalized to support security policies for integrity. Integrity policies [Biba 1977] are the dual of privacy policies. Just as privacy policies protect against data being *read* improperly, even if it passes through or is used by untrusted programs, integrity policies protect data from being improperly *written*. An integrity label keeps track of all the *sources* that have affected a value, even if those sources only affect the value indirectly. It prevents untrustworthy data from having an effect on trusted storage.

The structure of an *integrity policy* is identical to that of a privacy policy. It has an *owner*, the principal for whom the policy is enforced, and a set of *writers*: principals who are permitted to affect the data. An integrity label may contain a number of integrity policies with various owners. The intuitive meaning of an integrity policy is that it is a guarantee of quality. A policy $\{o : w_1, w_2\}$ is a guarantee by the principal o that only w_1 and w_2 will be able to affect the value of the data. The most restrictive integrity label is the label containing no policies, $\{\}$. This is the label that provides no guarantees as to the contents of the labeled value, and can be used as the data input only when the receiver imposes no integrity requirements. When a label contains multiple integrity policies, these policies are independent guarantees of quality from the owners of these policies.

Using an integrity label, a variable can be protected against improper modification. For example, suppose that a variable has a single policy $\{o : w_1, w_2\}$. A value labeled $\{o : w_1\}$ may be written to this variable, because that value has been affected only by w_1, and the label of the variable permits w_1 to affect it. If the value were labeled $\{o : w_1, w_3\}$, the write would not in general be permitted, because the value was affected by w_3, a principal not mentioned as an allowed writer in the label of the variable. (It would be permitted if $w_3 \succeq w_2$.) Finally, consider a value labeled $\{o : w_1;\ o' : w_3\}$. In this case, the write is permitted, because the first policy says that o believes only w_1 has affected the value. That the second policy exists on behalf of o' does not affect the legality of the write to the variable; it is a superfluous guarantee of quality.

Just as with privacy policies earlier, assignment relabels the value being copied into the variable, and to avoid violations of integrity, the label of the variable must be more restrictive than the label of the value. In Section 3.3, it was said that any legal relabeling for privacy policies is a sequence of incremental relabelings. This characterization is attractive because it is easier to judge the safety of an incremental relabeling than of a relabeling between two arbitrary labels. For an integrity label, there are also four kinds of safe incremental relabelings:

—**Add a writer**. This addition is safe because an additional writer in integrity policies is an additional contamination that can make the value only more restricted in subsequent use.

—**Remove a policy**. An integrity policy may be thought of as an assurance that at most the principals in a given set (the writers) have affected the data. Removing such an assurance is safe and restricts subsequent use of the value.

—**Replace a writer**. In a policy, a writer w' may be replaced by a writer w that it acts for. Because w' has the ability to act for w, a policy permitting w as a writer permits both w and w' as writers, whereas a policy permitting w' does not, in general, permit w. Therefore, replacing w' by w really adds writers, a change that is safe.

—**Add a policy**. The policy added, J, is identical to an existing policy I except that $\mathbf{o}(I) \succeq \mathbf{o}(J)$. The new policy offers a weaker integrity guarantee than the existing one, so the value becomes no less restrictive by the addition of this policy.

These four kinds of relabelings turn out to capture exactly the inverse of the relabelings that are allowed by the incremental rules for privacy labels, described in Section 3.3, because of these incremental relabeling steps can be reversed by applying one or two incremental relabeling steps for privacy. Similarly, each of the incremental privacy relabeling steps can be reversed by applying one or two of the incremental integrity relabeling steps. The implication of this observation is that the relabeling rule for integrity labels is precisely dual to the relabeling rule for privacy labels. For privacy labels L_1 and L_2 and corresponding integrity labels L'_1 and L'_2,

$$L_1 \sqsubseteq L_2 \longleftrightarrow L_2' \sqsubseteq L_1'$$

An analogue to declassification also exists for integrity labels. For privacy labels, the declassification mechanism allows privacy policies to be removed in cases where reasoning outside the scope of strict dependency analysis (as in the tax-preparer example) suggests that the policy is overly strict. The dual action for integrity policies is to *add* new integrity policies in situations where the data has higher integrity than strict dependency analysis might suggest. If a principal adds a new integrity policy to a label, or removes writers from an existing policy, it represents a statement of confidence in the integrity of the data, which allows that data to be used more freely subsequently. Just as with declassification for privacy, however, the reasons why a principal might choose to do so lie outside the scope of this model.

Adding new policies is safe because the new policy may be added only if the current process of the authority to act for the owner of the policy. Other principals will not be affected unless they trust the policy owner (and by extension, the process performing the declassification) to act for them.

Declassification can be described more formally: declassification of integrity label L_1 to a label L_2 is permitted when $L_2 \sqcap L_A^I \sqsubseteq L_1$, where L_A^I is an integrity label in which there is a policy for every principal in the authority of the process. Each such policy lists all principals in the system as writers. Note the duality of this rule to the rule in Section 3.6 for declassification of privacy labels. The symbol $\sqcap$ denotes the *meet* or greatest lower bound of the two labels, which is computed by taking the union of the integrity policies.

Integrity labels do introduce one new issue: code can damage integrity without access to any extra labeled resource. For example, the routine alleged to add two numbers might perform a different computation, destroying integrity. To keep track of this effect, an integrity label must be assigned to each function in a program, and joined with any value computed by the function. In a program expression like $f(x, y)$, all three sub-expressions (f, x, and y) have an associated integrity label.

Because an integrity label offers a quality guarantee, some authority is needed to label code with it—specifically, the authority to act for the owners of any integrity policies in the label. The owner of the code integrity label is not necessarily the author of the code. However, the author usually should appear as a writer in this integrity label.

Integrity and privacy labels both form lattices (or more precisely, pre-orders with distributive join and meet operations). The product of these two pre-orders is, similarly, a pre-order with distributive join and meet operations, so a system of combined labels can be defined that enforce privacy and integrity constraints simultaneously. The ordering relation for combined labels is defined in the usual way for products of ordered sets.

5. APPLICATION: THE Jif LANGUAGE

This section briefly describes both how the decentralized label model is applied to a real programming language, Jif, in which information flow can be checked statically; and also some novel features of Jif that make it a more practical programming language than previous languages with support for information flow control.

This work has involved several new contributions: Jif extends a complex programming language (Java) and supports many language features that previously have not been integrated with static flow checking, including mutable objects, subclassing, dynamic type tests, and exceptions. In fact, Jif appears to be the most expressive programming language for which static information flow control has been attempted. For example, because Jif has objects, it is at least as expressive as languages supporting first-class functions. Jif also provides powerful new features that make information flow checking less restrictive and more convenient than in previous programming languages:

—It provides a simple but powerful model of authority that allows code privileges to be checked statically, and also allows authority to be granted and checked dynamically.
—It provides *label polymorphism*, allowing code that is generic with respect to the security class of the data it manipulates.
—Run-time label checking and first-class label values provide a dynamic escape when static checking is too restrictive. Run-time checks are statically checked to ensure that information is not leaked by the success or failure of the run-time check itself.
—Automatic label inference makes it unnecessary to write many of the annotations that otherwise would be required.

Static analysis would be too limiting for structures like file systems, where information flow cannot be verified statically. Jif defines a new secure run-time escape hatch for these structures, with explicit run-time label checks. Uses of the run-time information flow mechanism are still partially verified statically, to ensure that they do not leak information.

5.1 Static checking and implicit flows

Incorporating the model in a programming language makes it easy for programmers to write code that is label correct and also enables compile time checking of information flow. The model can be incorporated into most strongly, statically-typed programming languages; Java was a natural choice because it is widely used, and its semantics are fairly well specified. The resulting language is called Jif, which stands for "Java Information Flow."

The idea behind Jif is that annotated programs are checked statically to verify that all information flows within the program follow the rules given in the previous section. Static information flow checking is, in fact, an extension to type checking. For both kinds of static analysis, the compiler determines that certain operations are not permitted to be performed on certain data values. The analogy goes further, because in a language with subtyping one can distinguish between notions of the apparent (static) and actual (run-time) type of a value. Similarly, static label checking can be considered as a conservative approximation to checking the actual run-time labels.

However, there is a difficulty in checking labels at run time that does not arise with type checking. Type checks may be performed at compile time or at run time, though compile-time checks are obviously preferred when applicable because they impose no run-time overhead.

By contrast, fine-grained information flow control is practical only with some static analysis, which may seem odd; after all, any check that can be performed by the compiler can be performed at run time as well. The difficulty with run-time checks is the fact that they can *fail*. In failing, or not failing, they may communicate information about the data that the program is running on. Unless the information flow model is properly constructed, the fact of failure (or its absence) can serve as a covert channel. By contrast, the failure of a compile-time check reveals no information about the actual data passing through a

```
x = 0;
if (b) {
    x = 1;
}
```

Fig. 6. Implicit information flow

program. A compile-time check only provides information about the program that is being compiled. Similarly, link-time and load-time checks provide information only about the program, and may be considered to be static checks for the purposes of this work.

Implicit information flow [Denning and Denning 1977] is difficult to prevent without static analysis. For example, consider the segment of code shown in Figure 6 and assume that the storage locations b and x belong to different security classes $\{b\}$ and $\{x\}$, respectively. (In the Jif language, a label component that is a variable name denotes all of the policies of the label of that variable, so $\{b\}$ means "the label of b".) In particular, assume b is more sensitive than x (more generally, $\{b\} \not\sqsubseteq \{x\}$), so data should not flow from b to x. However, the code segment stores 1 into x if b is true, and 0 into x if b is false; x effectively stores the value of b. A run-time check can detect easily that the assignment $x = 1$ communicates information improperly, and abort the program at this point. Consider, however, the case where b is false: no assignment to x occurs within the context in which b affects the flow of control. Knowledge of whether the program aborts or continues implicitly provides information about the value of b, which can be used at least in the case where b is false.

We could imagine inspecting the body of the if statement at run time to see whether it contains disallowed operations, but in general this requires evaluation of all possible execution paths of the program, which is clearly infeasible. Another possibility is to restrict all writes that follow the if statement on the grounds that once the process has observed b, it is irrevocably tainted. However, this approach seems too restrictive to be practical. The advantage of compile-time checking is that in effect, static analysis efficiently constructs proofs that *no* possible execution path contains disallowed operations.

Implicit flows are controlled in Jif by associating a static *program-counter label* ($\underline{pc}$) with every statement and expression; the program-counter label represents the information that might be learned from the knowledge that the statement or expression was evaluated. The label of an expression is always at least as restrictive as the program-counter label at the point of evaluation. Assume the initial $\underline{pc}$ in this example is the least restrictive label possible, $\{\}$. Then, inside the if statement conditioned on b, both $\underline{pc}$ and the label of the expression 1 are equal to $\{b\}$. At the assignment to x, the straightforward assignment rule of Section 3.3 enforces the desired requirement: $\{b\} \sqsubseteq \{x\}$.

5.2 Jif overview

In this section we sketch the essentials of the Jif language, which demonstrates that the information flow model can be applied practically to a rich computational model, providing sound and acceptably precise constraints on information flow. Because Jif is largely an extension to the well-known Java programming language [Gosling et al. 1996], this language description focuses on the differences between Jif and Java. A formal description of Jif is available, including the details of the static information flow analysis that is performed. [Myers 1999a; Myers 1999b].

This work has involved several new contributions. Because Jif extends a complex, object-oriented programming language, it supports many language features that have not been integrated with static flow checking previously, including mutable objects, subclassing, dynamic type tests, access control, and exceptions. Jif supports the usual Java types: integers and other ordinals, booleans, arrays, and objects, and the full range of Java statement forms. It supports class declarations, though not inner classes, and also supports exceptions. A few Java features, such as class variables, are not supported by Jif, and some minor changes are made to the semantics of exceptions. These changes prevent covert channels arising from the language definition, but do not substantially affect the expressive power of the language [Myers 1999a].

In Jif, every expression has a *labeled type* that consists of two parts: an ordinary Java type such as int, and a label that describes the ways that the value computed by the expression can propagate. The type and label parts of a labeled type act largely independently. Any type expression t may be labeled with any label expression $\{l\}$. This labeled type expression is written as $t\{l\}$; for example, the labeled type int{p: p} represents an integer that principal p owns and only p (and principals that can act for p) can read.

The goal of type checking is to ensure that the apparent, static type of each expression is a supertype of the actual, run-time type of every value it might produce; similarly, the goal of label checking is to ensure that the apparent label of every expression is at least as restrictive as the actual label of every value it might produce. In addition, label checking guarantees that, except when declassification is used, the apparent label of a value is at least as restrictive as the actual label of every value that might *affect* it. In principle, the actual label could be computed precisely at run time. Static checking ensures that the apparent, static label is always a conservative approximation of the actual label. For this reason, it is typically unnecessary to represent the actual label at run time.

The Jif language is extended with additional features that support information flow control:

—An explicit "declassify" operator allows the declassification of information.

—The "actsFor" statement performs a run-time test whether an acts-for relation exists between two principals. The information about existence of the acts-for relation is used during static checking of code in which it is known to exist.

—A call to a procedure may grant some of the authority possessed by the caller to the procedure being called, and the called procedure may test and use that authority.

—Variables and arguments may be declared to have the special type label, which permits run-time label checking. Variables of type label and argument-label parameters may be used to construct labels used in the code. Similarly, the first-class type principal allows run-time manipulation of information about the principal hierarchy, and variables of type principal may be used to construct run-time labels.

—A switch label statement can be used to determine the run-time labeled type of a value, and a special type protected conveniently encapsulates values along with their run-time labels.

In Jif, a label is denoted by a *label expression*, which is a set of *component expressions*. As in Section 3.2, a component expression of the form *owner: reader*$_1$, *reader*$_2$, ... denotes a policy. A label expression is a series of component expressions, separated by semicolons, such as $\{o_1\colon r_1, r_2;\ o_2\colon r_2, r_3\}$. In a program, a component expression

```
class passwordFile authority(root) {
  public boolean check (String user, String password)
   where authority(root) {
    // Return whether password is correct
    boolean match = false;
    try {
      for (int i = 0; i < names.length; i++) {
        if (names[i] == user &&
        passwords[i] == password) {
          match = true;
          break;
        }
      }
    } catch (NullPointerException e) {}
      catch (IndexOutOfBoundsException e) {}
    return declassify(match, {user; password});
  }
  private String [ ] names;
  private String { root: } [ ] passwords;
}
```

Fig. 7. A Jif password file

may take additional forms; for example, it may simply be a variable name. In that case, it denotes the set of policies in the label of that variable. The label {a} contains a single component; the meaning of the label is that the value it labels should be as restricted as the variable a is. The label {a; o: r} contains two components, indicating that the labeled value should be as restricted as a is, and also that the principal o restricts the value to be read by at most r.

5.3 Example: passwordFile

An example of a Jif program will illustrate how the language works. Figure 7 contains a Jif implementation of a simple password file, in which the passwords are protected by information flow controls. Only the method for checking passwords is shown. This method, check, accepts a password and a user name, and returns a boolean indicating whether the string is the right password for that user. In this method, the labels of the local variables match and i are not stated explicitly, and are automatically inferred from their uses.

The labels of the arguments passwords and user also are not stated explicitly and are implicit parameters to the method. This implicit label polymorphism allows the method to be called regardless of the labels of these two arguments. The actual labels of the arguments do not need to be known at run time, so there is no overhead associated with this construct. Label polymorphism is particularly important for building libraries of reusable code.

The if statement is conditional on the elements of passwords and on the variables user and password. Therefore, the body of the if statement has $\underline{pc}$ = {user; password; root:}, and the variable match also must have this label in order to allow the assignment match = true. This label prevents match from being returned directly as a result, because the label of the return value is the default label, {user; password}. Finally, the method declassifies match to this desired label, using its compiled-in authority to act for root.

More precise reasoning about the possibility of exceptions would make writing the code more convenient. In this example, the exceptions NullPointerException and IndexOutOf-

BoundsException must be caught explicitly, because the method does not explicitly declare them. It is feasible to prove in this case that the exceptions cannot be thrown, but the Jif compiler does not attempt to do so.

Otherwise there is very little difference between this code and the equivalent Java code. Only three annotations have been added: an authority clause stating that the principal root trusts the code, a declassify expression, and a label on the elements of passwords. The labels for all local variables and return values are either inferred automatically or assigned sensible defaults. The task of writing programs is made easier in Jif because label annotations tend to be required only where interesting security issues are present.

In this method, the implementor of the class has decided that declassification of match results in an acceptably small leak of information. Like all login procedures, this method does leak information, because exhaustively trying passwords eventually will extract the passwords from the password file. However, assuming that the space of passwords is large and passwords are difficult to guess, the expected amount of password information gained in each such trial is far less than one bit. Reasoning about when leaks of information are acceptable lies outside the domain of classic information flow control.

The declassify statement is checked statically. This declassification relabels data from the label $\{\mathsf{user};\mathsf{password};\mathsf{root}:\}$ to the label $\{\mathsf{user};\mathsf{password}\}$; that is, it removes a policy owned by the principal root. The statement is allowed because it is known statically at the point of invocation that the code acts with the privileges of this principal, because of the authority clauses on the method check and on its containing class. A class is required to declare its maximal authority, and different methods of the class may claim some or all of this authority. In order to add this class to the trusted execution platform as a piece of executable code, the permission of the principal root is required; otherwise, the authority of root could be acquired improperly.

5.4 Reasoning about the principal hierarchy

The static checker maintains a notion of the *static principal hierarchy* at every point in the program. The static principal hierarchy is a set of acts-for relations that are known to exist, and is a subset of the acts-for relations that exist in the principal hierarchy at run time.

The notion of the static hierarchy may be extended by testing principal hierarchy dynamically using the new actsFor statement. The statement $\mathsf{actsFor}(p_1, p_2)\ S$ executes the statement S if the principal p_1 can act for the principal p_2 in the current principal hierarchy. Otherwise, the statement S is skipped. Because the actsFor statement dynamically tests the authority of the principal p_1, it can be used as a simple access control mechanism, in addition to controlling the uses of declassification.

The statement S is checked statically using the knowledge that the tested acts-for relation exists: for example, if the static authority includes p_1, then during S it is augmented to include p_2. When compiling a statement of Jif code, the set of acts-for relations assumed to exist includes all those relations indicated by containing actsFor statements. Because an acts-for relation is assumed to exist at compile time only in a context in which an actsFor test has ensured at run time that it does exist, the dynamic, run-time principal hierarchy may differ from the static principal hierarchy only by the addition of more acts-for relations. The information about the principal hierarchy that is available at the time of the check may be partial: any acts-for relations observed must exist at run time; however, additional acts-for relations and even additional principals may exist at run time when the relabeling is performed.

With this assumption, we can see why it was necessary in Section 3.4 that each policy in the label L_1 be covered by some single policy in L_2 in order for the relabeling from L_1 to L_2 to be safe. It might seem that a relabeling would be safe if, for each policy I in L_1, there were some *set* of policies J_i in L_2 that cover I, in the sense that for all i, $\mathbf{o}(J_i) \succeq \mathbf{o}(I)$ and $\bigcap_i \mathbf{R}(J_i) \subseteq \mathbf{R}(I)$. However, if no individual policy covers I, it is always possible in this situation to add new principals and acts-for relations to the principal hierarchy in such a way that the set of policies J_i no longer cover the I. Because each member of the set does not cover I by itself, each member must contain at least one principal p_i that is not in $\mathbf{R}(I)$. Let us now add another principal p' to the principal hierarchy that can act for all of the p_i. However, all of the policies J_i now implicitly allow p' as a reader, whereas I does not. Therefore, the relabeling from L_1 to L_2 clearly is not safe because the policies J_i do not cover I even when considered jointly in this extended principal hierarchy.

6. 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[Bell and LaPadula 1975] and Denning [Denning 1976]. A commonly used label model fitting into the lattice formalism is that of Feiertag et al. [Feiertag et al. 1977].

The decentralized label model has several similarities to the ORAC model of McCollum et al. [McCollum et al. 1990]; both models provide some approximation of the "originator-controlled release" labeling used by the U.S. DoD/Intelligence community. Both ORAC and the decentralized label model have the key concept of *ownership* of policies. Both models also support the joining of labels as computation occurs, though the ORAC model lacks some important lattice properties since it attempts to merge policies with common owners. In the ORAC model, as in some mandatory access control models, both process labels and object labels can float upward in the label lattice arbitrarily, a phenomenon called *label creep* that leads to excessively restrictive labels. The absence of lattice properties and the dynamic binding of labels to objects and processes makes any static analysis of the ORAC model rather difficult. Interestingly, ORAC does allow owners to be replaced in label components (based on ACL checks that are analogous to acts-for checks), but it does not support extension of the reader set. The ORAC model also does not support any form of declassification.

Declassification in most information flow control systems lies outside the model. One recent model by Ferrari et al. [Ferrari et al. 1997] incorporates 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. Also, Abadi [Abadi 1997] 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 declassification.

Biba showed that information flow control can be used to enforce *integrity* as well as secrecy, and that integrity is a dual of secrecy [Biba 1977]; this insight has been employed in several subsequent systems, and also applies to the decentralized integrity policies described in Section 4, which extend earlier integrity models with a richer set of relabelings and a notion of selective declassification.

One important property of the decentralized label model is that it is amenable to static checking of information flow. However, dynamic enforcement is the approach described in the Orange Book [Department of Defense 1985] and most systems have followed this approach, leading to difficulties in controlling implicit flows (for example, [Fenton 1973; Fenton 1974; McIlroy and Reeds 1992]). A dynamic model integrating both access control and information flow control, defined formally using denotational semantics, was developed but not implemented by Stoughton [Stoughton 1981].

Static analysis of information flow has a long history, although it has not been as widely used as dynamic checking. Denning originally proposed a language to permit static checking [Denning and Denning 1977], but it was not implemented. Another approach to checking programs for information flows statically has been automatic or semi-automatic theorem proving. Researchers at MITRE [Millen 1976; Millen 1981] and SRI [Feiertag 1980] developed techniques for information flow checking using formal specifications. Feiertag [Feiertag 1980] developed a tool for automatically checking these specifications using a Boyer-Moore theorem prover.

Recently, there has been more interest in provably-secure programming languages, treating information flow checks in the domain of type checking, which does not require a theorem prover. A simple type system for checking integrity has been developed [Palsberg and Ørbæk 1995]. A similar approach has been taken to static analysis of secrecy, encoding Denning's rules in simple languages and showing them to be sound using standard programming language techniques [Volpano et al. 1996]. Security labels have also been applied to a simple functional language with reference types (the SLam calculus), in which an integrated model similar to that of Stoughton is shown to be statically checkable and to enforce noninterference [Heintze and Riecke 1998]. An extension of this type-based approach as a generalized dependency analysis builds on the SLam calculus [Abadi et al. 1999].

These recent programming-language approaches have the limitation that they are entirely static: unlike Jif, they have no run-time access control, no declassification, and no run-time flow checking. These models also do not provide label polymorphism, support for objects, or tracking of fine-grained information flow through exceptions. The addition of these features in Jif is important for supporting a realistic programming model. However, these features do make Jif more difficult to treat with the current formal tools of programming language theory.

Recent models for information flow have provided various definitions of when information flow is considered to occur. These models include non-interference [Goguen and Meseguer 1982], non-deducibility and other possibilistic extensions of the non-interference property [Sutherland 1986; McCullough 1987; McLean 1994; Zakinthinos and Lee 1997], and probabilistic and information-theoretic approaches [Gray 1990; McLean 1990; Gray 1991]. Focardi and Gorrieri have developed a tool for precise automatic checking of information flow with respect to a variety of information flow models in concurrent programs; however, the cost of precise analysis is that this methodology does not scale to programs of a realistic size. [Focardi and Gorrieri 1997].

The notion of security enforced by Jif is essentially the same as in the other language-based work on information flow: non-interference. Intuitively, non-interference requires that the low-security outputs of a program may not be affected by its high-security inputs. In Jif, high-security inputs may affect low-security outputs, but only if there is a declassify operation along some path of information flow from the inputs to the outputs. Thus, if a principal p does not grant its authority to any running code, non-interference is enforced relative to that principal; considering only the components of a label owned by p (and principals that can act for p), program outputs are not affected by a program inputs with less restrictive labels. Other recent work has explored the notion of selective declassification introduced by Jif, in which a primitive access control mechanism is used to selectively mediate declassification; this work also coins the term “selective declassification” [Pottier and Conchon 2000].

Jif does not consider covert channels arising from the measurement of time and asynchronous communication between threads. A scheme for statically analyzing thread communication has been proposed [Andrews and Reitman 1980]; essentially, a second pc label is added with different propagation rules. This second label is used to restrict communication with other threads. The same technique would address many timing channels, and could be applied to Jif. However, this technique seems likely to impractically restrict multi-threaded programs. Smith and Volpano have developed rules recently for checking information flow in a multi-threaded functional language [Smith and Volpano 1998]. The rules they define, while proven sound, prevent the run time of a program from depending in any way on non-public data, which is arguably impractical. Programs can be transformed to eliminate some classes of timing channels [Agat 2000], but only in a simple imperative language lacking functions.

7. CONCLUSIONS

Protecting privacy and secrecy of data has long been known to be a very difficult problem. The increasing use of untrusted programs in decentralized environments with mutual distrust makes a solution to this problem both more important and more difficult to solve. Existing security techniques do not provide satisfactory solutions to this problem.

The goal of this work is to make information flow control a viable technique for providing privacy in a complex, decentralized world with mutually distrusting principals. Information flow control is an attractive approach to protecting the privacy (and integrity) of data because it allows security requirements to be extended transitively towards or away from the principals whose security is being protected. However, it has not been a widely accepted technique because of the excessive restrictiveness it imposes and the computational overhead.

To address these limitations of conventional information flow techniques, this work addresses two separable problems. This paper has concentrated on the first, providing an expressive model of decentralized information flow labels with the ability to express privacy policies for multiple, mutually distrusting principals, and to enforce all of their security requirements simultaneously. A second part of the work that has only been sketched by this paper is the new language Jif, which permits static checking of decentralized information flow annotations. Jif seems to be the most practical programming language yet that allows this checking.

8. FUTURE WORK

There are several directions for extending this work. One obviously important direction is to continue to make Jif a more practical system for writing applications. Jif addresses many of the limitations of earlier information flow systems that have prevented their use for the development of reasonable applications; however, more experience is needed to better understand the practical applications of this approach.

One direction for exploration is the development of secure run-time libraries written in Jif that support Jif applications. Features of Jif such as polymorphism and hybrid static/dynamic checking should make it possible to write such libraries in a generic and reusable fashion.

It should also be possible to augment the Java Virtual Machine [Lindholm and Yellin 1996] with annotations similar to those used in Jif source code. The bytecode verifier would check both types and labels at the time that code is downloaded into the system. Other recent work [Lindholm and Yellin 1996; Necula 1997; Morrisett et al. 1998] has shown that type checking performed at compile time can be transformed into machine-code or bytecode annotations. The code can then be transmitted along with the annotations, and the two checked by their receiver to ensure that the machine code obeys the constraints established at compile time. This approach also should be applicable to information flow annotations that are expressible as a type system.

The Jif language contains complex features such as objects, inheritance and dependent types, and these features have made it difficult thus far to use standard programming-language proof techniques to show that the language is sound with respect to an information flow model such as non-interference. However, this demonstration is important for widespread acceptance of a language for secure computation and seems likely to be feasible in the near term.

This work has assumed an entirely trusted execution environment, but future computing environments will be large, networked systems in which different principals may have different levels of trust in the various hosts in the network. It seems likely that the label model will extend in a natural manner to a such an environment, because of the decentralized structure of the label model and its support for reasoning about labels with only partial information about the principal hierarchy.

This work shows how to control several kinds of information flow channels better, including channels through storage, implicit flows, and run-time security checks. However, covert channels that arise from timing channels and from the timing of asynchronous communication between threads are avoided in Jif at present by ruling out timing and multi-threaded code. Supporting multi-threaded applications would make this work more widely applicable. Although there has been work on analyzing and controlling these channels with language-based techniques, [Smith and Volpano 1998; Heintze and Riecke 1998; Agat 2000], the current techniques are limited in applicability.

ACKNOWLEDGMENTS

Many people have helped improve this work. Among those making insightful suggestions were Martín Abadi, Atul Adya, Kavita Bala, Miguel Castro, Stephen Garland, Butler Lampson, Fred Schneider, and the anonymous reviewers. Nick Mathewson and Steve Zdancewic helped develop much of the code that ended up in the Jif compiler.

REFERENCES

ABADI, M. 1997. Secrecy by typing in security protocols. In *Proc. Theoretical Aspects of Computer Software: Third International Conference* (Sept. 1997).

ABADI, M., BANERJEE, A., HEINTZE, N., AND RIECKE, J. 1999. A core calculus of dependency. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)* (San Antonio, TX, USA, Jan. 1999), pp. 147–160.

AGAT, J. 2000. Transforming out timing leaks. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)* (Jan. 2000).

ANDREWS, G. R. AND REITMAN, R. P. 1980. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems 2*, 1, 56–76.

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

BIBA, K. J. 1977. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372 (April), USAF Electronic Systems Division, Bedford, MA.

DENNING, D. E. 1976. A lattice model of secure information flow. *Comm. of the ACM 19*, 5, 236–243.

DENNING, D. E. AND DENNING, P. J. 1977. Certification of Programs for Secure Information Flow. *Comm. of the ACM 20*, 7 (July), 504–513.

Department of Defense. 1985. *Department of Defense Trusted Computer System Evaluation Criteria* (DOD 5200.28-STD (The Orange Book) ed.). Department of Defense.

FEIERTAG, R. J. 1980. A technique for proving specifications are multilevel secure. Technical Report CSL-109 (Jan.), SRI International Computer Science Lab, Menlo Park, California.

FEIERTAG, R. J., LEVITT, K. N., AND ROBINSON, L. 1977. Proving multilevel security of a system design. *Proc. 6th ACM Symp. on Operating System Principles (SOSP), ACM Operating Systems Review 11*, 5 (Nov.), 57–66.

FENTON, J. S. 1973. *Information Protection Systems*. Ph. D. thesis, University of Cambridge, Cambridge, England.

FENTON, J. S. 1974. Memoryless subsystems. *Computing J. 17*, 2 (May), 143–147.

FERRARI, E., SAMARATI, P., BERTINO, E., AND JAJODIA, S. 1997. Providing flexibility in information flow control for object-oriented systems. In *Proc. IEEE Symposium on Security and Privacy* (Oakland, CA, USA, May 1997), pp. 130–140.

FOCARDI, R. AND GORRIERI, R. 1997. The compositional security checker: A tool for the verification of information flow security properties. *IEEE Transactions on Software Engineering 23*, 9 (Sept.), 550–571.

GOGUEN, J. A. AND MESEGUER, J. 1982. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy* (April 1982), pp. 11–20.

GOGUEN, J. A. AND MESEGUER, J. 1984. Unwinding and inference control. In *Proc. IEEE Symposium on Security and Privacy* (April 1984), pp. 75–86.

GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java Language Specification*. Addison-Wesley. ISBN 0-201-63451-1.

GRAY, J. W., III. 1990. Probabilistic interference. In *Proc. IEEE Symposium on Security and Privacy* (May 1990), pp. 170–179.

GRAY, J. W., III. 1991. Towards a mathematical foundation for information flow security. In *Proc. IEEE Symposium on Security and Privacy* (1991), pp. 21–34.

HEINTZE, N. AND RIECKE, J. G. 1998. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)* (San Diego, California, Jan. 1998).

KANG, M. H., MOSKOWITZ, I. S., AND LEE, D. C. 1995. A network version of the pump. In *Proc. IEEE Symposium on Security and Privacy* (Oakland, CA, May 1995).

KARGER, P. A. AND WRAY, J. C. 1991. Storage channels in disk arm optimization. In *Proc. IEEE Symposium on Security and Privacy* (Oakland, CA, May 1991).

LAMPSON, B., ABADI, M., BURROWS, M., AND WOBBER, E. 1991. Authentication in distributed systems: Theory and practice. In *Proc. 13th ACM Symp. on Operating System Principles (SOSP)* (October 1991), pp. 165–182. *Operating System Review*, 253(5).

LINDHOLM, T. AND YELLIN, F. 1996. *The Java Virtual Machine*. Addison-Wesley, Englewood Cliffs, NJ.

MCCOLLUM, C. J., MESSING, J. R., AND NOTARGIACOMO, L. 1990. Beyond the pale of MAC and DAC—defining new forms of access control. In *Proc. IEEE Symposium on Security and Privacy* (1990), pp. 190–200.

MCCULLOUGH, D. 1987. Specifications for multi-level security and a hook-up property. In *Proc. IEEE Symposium on Security and Privacy* (May 1987). IEEE Press.

MCILROY, M. D. AND REEDS, J. A. 1992. Multilevel security in the UNIX tradition. *Software—Practice and Experience 22*, 8 (Aug.), 673–694.

MCLEAN, J. 1990. Security models and information flow. In *Proc. IEEE Symposium on Security and Privacy* (1990), pp. 180–187.

McLean, J. 1994. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symposium on Security and Privacy* (May 1994), pp. 79–93.

Millen, J. K. 1976. Security kernel validation in practice. *Comm. of the ACM 19*, 5 (May), 243–250.

Millen, J. K. 1981. Information flow analysis of formal specifications. In *Proc. IEEE Symposium on Security and Privacy* (April 1981), pp. 3–8.

Millen, J. K. 1987. Covert channel capacity. In *Proc. IEEE Symposium on Security and Privacy* (Oakland, CA, 1987).

Morrisett, G., Walker, D., Crary, K., and Glew, N. 1998. From System F to typed assembly language. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)* (San Diego, California, Jan. 1998).

Myers, A. C. 1999a. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)* (San Antonio, TX, USA, Jan. 1999).

Myers, A. C. 1999b. *Mostly-Static Decentralized Information Flow Control*. Ph. D. thesis, Massachusetts Institute of Technology, Cambridge, MA.

Myers, A. C. and Liskov, B. 1998. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symposium on Security and Privacy* (Oakland, CA, USA, May 1998).

Necula, G. C. 1997. Proof-carrying code. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL)* (Jan. 1997), pp. 106–119.

Palsberg, J. and Ørbæk, P. 1995. Trust in the λ-calculus. In *Proc. 2nd International Symposium on Static Analysis*, Number 983 in Lecture Notes in Computer Science (Sept. 1995), pp. 314–329. Springer.

Pottier, F. and Conchon, S. 2000. Information flow inference for free. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)* (Sept. 2000).

Sandhu, R. S. 1996. Role hierarchies and constraints for lattice-based access controls. In *Proc. Fourth European Symposium on Research in Computer Security* (Rome, Italy, September 25-27 1996).

Smith, G. and Volpano, D. 1998. Secure information flow in a multi-threaded imperative language. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)* (San Diego, California, Jan. 1998).

Stoughton, A. 1981. Access flow: A protection model which integrates access control and information flow. In *IEEE Symposium on Security and Privacy* (1981), pp. 9–18. IEEE Computer Society Press.

Sutherland, D. 1986. A model of information. In *Proc. 9th National Security Conference* (Gaithersburg, Md., 1986), pp. 175–183.

Volpano, D., Smith, G., and Irvine, C. 1996. A sound type system for secure flow analysis. *Journal of Computer Security 4*, 3, 167–187.

Wittbold, J. T. and Johnson, D. M. 1990. Information flow in nondeterministic systems. In *Proc. IEEE Symposium on Security and Privacy* (May 1990), pp. 144–161.

Zakinthinos, A. and Lee, E. S. 1997. A general theory of security properties and secure composition. In *Proc. IEEE Symposium on Security and Privacy* (Oakland, CA, 1997).