Proceedings of the 16th ACM Symposium on Operating Systems Principles, Saint-Malo, France, October 1997

A Decentralized Model for Information Flow Control

Andrew C. Myers **Barbara Liskov**

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

Abstract

This paper presents a new model for controlling information flow in systems with mutual distrust and decentralized authority. The model allows users to share information with distrusted code (e.g., downloaded applets), yet still control how that code disseminates the shared information to others. 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. The paper also shows how static program analysis can be used to certify proper information flows in this model and to avoid most run-time information flow checks.

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. For example, if user A is allowed to read B's data, B cannot control how A distributes the information it has read. Control of information propagation *is* supported by existing information flow and compartmental models, but these models unduly restrict the computation that can be performed. The goal of this work is to make information flow control more useful by relaxing these restrictions.

Information flow control is vital for large or extensible systems. In a small system, preventing improper propagation of information is easy: you don't pass data to code whose implementation is not completely trusted. This simple rule breaks down in larger systems, because the trust requirement is transitive: any code the data might travel to must also be trusted, requiring complete understanding of the code. As the system grows larger and more complex, and incorporates distrusted code (e.g., web applications), complete trust becomes unattainable.

Systems that support the downloading of distrusted code are particularly in need of a better security model. For example, Java [GJS96] supports downloading of code from remote sites, which creates the possibility that the downloaded code will transfer private data to those sites. Java attempts to prevent these transfers by using a compartmental security model, but this approach largely prevents applications from sharing data. Also, different data manipulated by an application have different security requirements. A security model is needed that supports fine-grained information sharing between distrusted applications, while reducing the potential for information leaks.

This paper contains exploratory work towards a new model of decentralized information flow that is also inexpensive in both space and time. Our model allows users to control the flow of their information without imposing the rigid constraints of a traditional multilevel security system. The goal of this model is to provide security guarantees to users and to groups rather than to a monolithic organization. It differs from previous work on information flow control by allowing users to explicitly *declassify* (or *downgrade*) data that they own. When data is derived from several sources, all the sources must agree to release the data.

Our model can largely be checked statically, like some existing information flow models [DD77, AR80]. We define user-supplied program annotations, called *labels*, that describe the allowed flow of information in a program. An-

This research was supported in part by DARPA Contract N00014-91-J-4136, monitored by the Office of Naval Research, and in part by DARPA Contract F30602-96-C-0303, monitored by USAF Rome Laboratory.

notated programs can be checked at compile time, in a manner similar to type checking, to ensure that they do not violate information flow rules. Compile-time checks have no run-time overhead in space or time, and unlike run-time checks, when they fail, they do not leak information about the data the program is using.

Our work extends existing models by allowing individuals to declassify data they own, rather than requiring a central authority to do it. In addition, we extend the static checking model in three ways. First, we introduce an implicit form of parametric polymorphism, called *label polymorphism*, to express procedures that are parametric with respect to the security labels of their arguments, and with respect to the principal on whose behalf the procedure executes. Label polymorphism extends the power of static analysis, and allows programmers to write generic code. Second, since purely static analysis would be too limiting for structures like file systems, where information flow cannot be verified statically, we define 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. Third, we show that despite these features, the labels of local variables can be inferred automatically, easing the job of adding flow annotations to a program.

Our goal in exploring these techniques is to eventually support the following useful applications:

- Secure servers and other heavily-used applications can be written in programming languages extended with information flow annotations, adding confidence that sensitive information is not revealed to clients of the service through programming errors.
- Secure compiled code may be transferred from a remote site and run locally with less concern that it might leak information. Code transfer is useful both for clients, which download applications from servers, and for servers, which upload code and data from clients for remote evaluation.

The annotations could be used to extend many conventional programming languages, intermediate code (such as JVM [LY96]), or machine code, where the labeling system defined here makes a good basis for security proofs [Nec97]. Labeled machine code and security proofs could work together: proof annotations for object code would be generated as a byproduct of compiling a program that contains information flow annotations.

The remainder of the paper describes the model and how checking is done. The model is intended to control covert and legitimate storage channels; it does not deal with timing channels, which are harder to control. The work assumes the existence of a reliable, efficient authentication mechanism, and of a trusted execution platform; for example, code may be executed by a trusted interpreter, or generated only by a trusted compiler. When the computational environment contains many trusted nodes connected by a network, the communication links beween the nodes must be trusted, which can be accomplished by encrypting communication between nodes.

The organization of the remainder of this 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 discusses issues that arise when code using the new model is statically checked for correctness. Section 5 shows how the model can be integrated into a simple programming language. Section 6 shows how to infer most labels in programs automatically, making the job of annotating a program much simpler. Section 7 describes related work in the areas of information flow models, access control, and static program analysis. We conclude in Section 8 and discuss future work in Section 9.

2 Motivating Examples

Let us consider two examples for which a decentralized model of information flow is helpful — the *medical study* and the *bank*, depicted in Figures 1 and 2. The scenarios place somewhat different demands on the information flow model. They demonstrate that our approach permits legitimate flows that would not be allowed with conventional information flow control, and that it is easy to determine that information is not being leaked.

In the figures, an oval represents a principal within the system, and is labeled with a boldface character that indicates the authority with which it acts. For example, in the medical study (Figure 1), the important principals are the patient, **p**, a group of researchers, **R**, the owners of a statistical analysis package, **S**, and a trusted agent, **E**. Arrows in the diagrams represent information flows between principals; square boxes represent information that is flowing, or databases of some sort.

Each principal can independently specify policies for the propagation of its information. These policies are indicated by labels of the form $\{O : R\}$, meaning that *owner* O allows the information to be read by readers R, where O is a principal and R is a set of principals. The owner is the source of the information and has the ability to control the policy for its use. For example, in the medical study example, the patient's medical history may be read only by principals with the authority to act on behalf of either the patient principal p or the hospital principal H.

In the diagrams, double ovals represent trusted agents that declassify information (for example, **E** in the medical study). These agents have the authority to act on behalf

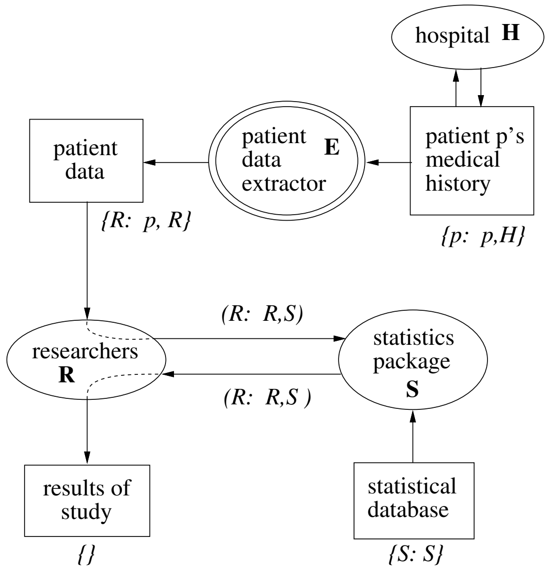

Figure 1: Medical Study Scenario

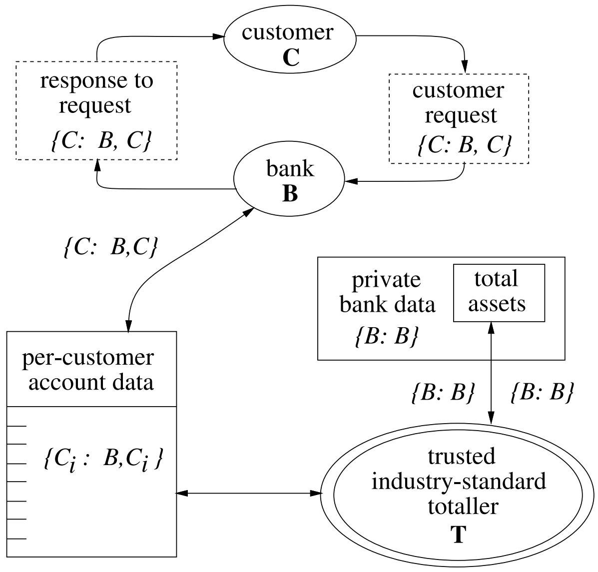

Figure 2: Bank Scenario

of a principal in the system, and may therefore modify the policies that have been attached to data by that principal. One goal of these two examples is show how our approach limits the trust that is needed by participants in the system; the double ovals identify the places where special trust is needed.

2.1 The Medical Study

The medical study example shows that it is possible to give another party private information and receive the results of its computation while remaining confident that the data given to it is not leaked. The purpose of the study is to perform a statistical analysis of the medical records of a large number of patients. Obviously, the patients would like to keep specific details of their medical history private. The patients give permission to the researchers performing the study to use their medical data to produce statistics, with the understanding that their names and other identifying information will not be released. Thus, the patients put some trust in the *patient data extractor*, **E**, which delivers to the researchers a suitably abridged version of the patient records. The data extractor has the authority to act for the patient (**p**), so it can replace the patient's policy $\{p\text{: } p,H\}$ with the researcher-controlled policy, $\{R\text{: } p,R\}$, which allows the extracted data to be read by the researchers and by the patient.

The researchers would like to use a statistical analysis package that they have obtained from another source, but the patients and researchers want the guarantee that the analysis package will not leak their data to a third party. To accomplish this, the researchers relabel the patient data with $\{R\text{: } R,S\}$. The analysis package is able to observe but not to leak the relabeled data since S is only a reader, not an owner.

The analysis package performs its computations, using the patient data, now labeled $\{R\text{: } R,S\}$, and its own statistical database, labeled $\{S\text{: } S\}$. The writers of the analysis package would also like some assurance that their statistical database is not being leaked to the researchers. The result of the computation must retain the policies of both R and S, and therefore acquires the joint label $\{R\text{: } R,S;\ S\text{: } S\}$. This label only allows flows to the principal S, since S is the only principal in both reader sets. The analysis package then explicitly declassifies the result of the computation, changing the label to $\{R\text{: } R,S\}$ so the researchers can read it. Note that since the analysis package can declassify the analysis result, it is not forced to declassify all information extracted from the statistical database, which would probably require more careful analysis of the analysis code to show that the database contents were not leaked.

Finally, the researchers may declassify the result of their study, changing the label $\{R\text{: } R,S\}$ to the unrestricted label $\{\}$. This change allows the general public to see their results, and is acceptable as long as there are so many patients in the study that information about individual patients cannot be extracted from the final result.

This example uses declassification in four places. Each time, declassification takes place according to the simple rule that a principal may modify its own flow policies. Conventional information flow control has no notion of declassification within the label system, and therefore, cannot model this example.

2.2 The Bank

The bank scenario is illustrated in Figure 2. A bank serves many customers, each of whom would like to keep his data safe from other customers and non-customers. In addition, the bank stores private information, such as its current assets and investments, that it would like to keep safe from all customers and non-customers.

The bank receives periodic requests from each customer, e.g., to withdraw or deposit money. Each request should be able to observe only information that is owned by that customer, and none of the bank's private data. The bank is better than real banks in that it allows customers to control dissemination of their account information; each customer has a distinct information flow policy for his account information, which prevents the bank from leaking the information to another party. The customer's request, the account itself, and the bank's response to the request are all labeled $\{C\text{: }B,C\}$, allowing the bank to read the information but not to control it. However, the bank's private database, including its record of total assets, is most naturally labeled $\{B\text{: }B\}$.

To keep the total assets up to date, information derived from the customer's request must be applied to the total assets. To make this possible, the customer places trust in the *totaller*, *T*, a small piece of the bank software that acts with the authority of both the customer and the bank, and therefore can declassify the amount of the customer request in order to apply it to the total asset record. Conceivably, the totaller is a certified, industry-standard component that the customer trusts more than the rest of the bank software. Another reasonable model is that the totaller is part of an audit facility that is outside the bank's control.

3 Decentralized Information Flow Control

This section describes our new model of decentralized information flow. The model assumes a set of *principals* representing users and other authority entities. To avoid loss of generality, a process has the authority to act on behalf of some set of principals.

Computations manipulate *values*. Values are obtained from *slots*—variables, objects, other storage locations—that can serve as sources and sinks for values, and from computations; values can also be obtained from *input channels*, which are read-only slots that allow information to enter the system. A value can be written either to a slot or to an *output channel*, which serves as an information sink that transmits data outside the system.

Values, slots, and channels all have attached *labels*, which are a more flexible form of the security classes encountered in most information flow models. The flexibility introduced by our labels makes decentralized declassification possible.

The label on a value cannot change, but a new copy of the value can be created with a new label. When this happens we say the value is *relabeled*, though it is really only the copy that has a new label. The key to secure flow is to ensure that any relabeling is consistent with the security policies of the original labeling. Only values can be relabeled; slots and channels cannot. This restriction allows us to check information flows at compile time. If either slots or channels could be relabeled, we would need run-time label checks whenever they were used.

Sections 3.1 and 3.2 present our new model. Section 3.3 discusses principals in more detail and explains how we achieve flexibility even though slots and channels cannot be relabeled. Section 3.4 defines the relabeling rules, Section 3.5 explains output channels further, and Section 3.6 discusses how our model forms a conventional security-class lattice.

3.1 Overview

A label L contains a set of principals called the *owner set*, or *owners*(L). The owners are the principals whose data was observed in order to construct the data value; they are the original *sources* of the information. For each owner O, the label also contains a set of principals called the *reader set*, or *readers*(L,O). The reader set for a particular owner specifies the principals to whom the owner is willing to release the value. Together, the *owners* and *readers* functions completely specify the contents of a label. A useful concept is the *effective reader set* of L: the set of principals that all owners of the data agree to allow to release it to. The *effective reader set* is the intersection of every reader set in L.

An example of an expression that denotes a label L is the following: $\{o1\text{: }r1,r2;\ o2\text{: }r2,r3\}$, where $o1$, $o2$, $r1$, $r2$ denote principals. The owners of this label are $o1$ and $o2$, the reader sets for these owners are *readers*$(L, o1) = \{r1, r2\}$ and *readers*$(L, o2) = \{r2, r3\}$, and the effective reader set is $\{r2\}$.

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 the flow policy for the owner's part of the label; in particular, it 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. The labels maintain independent reader sets for each owning principal. If, instead, a label consisted of just an owner set and a reader set, we would lose information about the individual flow policies of the owners and reduce the power of declassification.

The key to controlling information flow is to ensure that the policies of each owner are enforced as data is read and written. However, when a value is read from a slot, it acquires the slot's label, which means that whatever label that value had at the time it was written to the slot is no longer known when it is read. In other words, writing a value to a slot is a relabeling. This loss of information is acceptable provided there is no loss of control.

Therefore, we allow a value to be assigned to a slot only if the relabeling that occurs at this point is a *restriction*, a relabeling in which the new label allows fewer accesses than the original; this happens, for example, if the slot's label allows fewer readers for an owner than the value's label. A relabeling from label L_1 to label L_2 is a restriction, written $L_1 \sqsubseteq L_2$, if L_1 has more readers and fewer owners than L_2:

Definition of $L_1 \sqsubseteq L_2$

$$\begin{array}{rcl} owners(L_1) & \subseteq & owners(L_2) \\ \forall O \in owners(L_1), readers(L_1, O) & \supseteq & readers(L_2, O) \end{array}$$

Note that the rules for readers and owners are opposites.

We could have used a different model in which a slot stores both a value and the label of that value. However, in that model the label that is stored in the slot becomes another information channel. Also, this model would not permit compile-time label checking. Our approach does permit this checking, and therefore labels cause little run-time overhead. As discussed in Section 5, labels can be values themselves; this feature allows label checking to be deferred till run time, overcoming the limitations of doing all checking at compile time.

In addition to slots, the system contains channels, which allow interaction with external devices: *input channels* allow information to be read and *output channels* allow information to be written. Reading from an input channel is just like reading from a slot; the value is given the channel's label. However, writing to an output channel is different from writing to a slot; as discussed further in Section 3.5, writing to an output channel is legal if the channel's readers are a subset of the readers allowed by the data being written. Creation of new channels is obviously a sensitive operation.

In this model, it is safe for a process to manipulate data even though the current principal does not have the right to read it. This follows because all the process can do with the data is write it to a slot or a channel provided the data's label allows this. Thus, access-control read checks aren't needed! Nevertheless, such checks might be desired, to reduce the risk of exposure through covert channels of sensitive data such as passwords. One possible extension would be to fold read access checks into label checking: a process can read information from a slot with label L only if the process can act for some principal R in the effective reader set of L.

3.2 Derived Labels

During computation, values are derived from other values. Since a derived value may contain information about its sources, its label must reflect the policies of each of its sources. For example, if we multiply two integers, the product's label must reflect the labels of both operands.

When a program combines two values labeled with L_1 and L_2, respectively, the result should have the least restrictive label that maintains all the flow restrictions specified by L_1 and L_2. This least restrictive label, the *join* of L_1 and L_2 (written as $L_1 \sqcup L_2$), is constructed as follows: The owner set of $L_1 \sqcup L_2$ is the union of the owner sets of L_1 and L_2, and the reader set for each owner in L_1 and L_2 is the intersection of their corresponding reader sets. This rule can be written concisely, assuming the following natural definition: *readers*(L, O) for an O that is *not* in the owner set is defined to be the set of all principals, since O imposes no restrictions on propagation. The join rule is then the following:

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

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

(The symbol $\oplus$ has also been used to denote the join of two security classes [DD77, AR80].)

Note that $L_1 \sqsubseteq L_1 \sqcup L_2$ for all labels L_1 and L_2. Joining is a restriction and therefore it does not leak information.

3.3 The Principal Hierarchy

To allow compile-time analysis, slots must be immutably labeled. Immutable slot labels might seem like a limitation, but we provide two mechanisms to make the labels on slots more flexible: run-time labeling, which is discussed later, and modification of the rights of principals, which changes the set of data that principals can read.

Within a system, principals serve various functions: some represent the full authority of a user of the system; others represent groups of users; and still others represent roles, restricted forms of a user's authority. In practice, these different principals are used quite differently, and many systems treat them as entirely different entities. Some principals have the right to *act for* other principals and assume their power. For example, every member of a group might have the right to act for the group principal. The *acts for* relation is reflexive and transitive, defining a hierarchy or partial order of principals. This model is similar to a *speaks for* hierarchy [LABW91], though roles are treated here as first-class principals. We assume that the principal structure can be queried using the primitive *acts-for* function to discover whether the current principal has the right to act for another principal.

The right of one principal to act for another is recorded in a database. The database can be modified: for example, to alter the membership of groups, or to transfer a role from one employee to another. Obviously, modifications to the principal structure are extremely powerful and must be restricted by some form of access control. Also, to prevent modifications to the principal structure from serving as a covert channel, the principal database must be labeled in a way that prevents information leaks, just as ordinary databases in the system must be.

3.4 Relabeling Rules

This section restates and discusses our two relabeling rules, restriction, which defines the legality of assignment, and declassification, which allows an owner to modify its flow policy:

Rule 1: Restriction. A relabeling from L_1 to L_2 is valid if it is a restriction: $L_1 \sqsubseteq L_2$. Intuitively, it removes readers, adds owners, or both.

Rule 2: Declassification. A declassification either adds readers for some owner O or removes the owner O. This relabeling can be done only if the process acts for O.

Relabeling by restriction is independent of the principal hierarchy and requires no special privilege to perform. *Declassification*, by contrast, depends on the acts-for relation and is legal only when the current process possesses the needed authority. It introduces potential security leaks, but only leaks of information owned by the current principal. The current principal has the power to leak its own information, but cannot leak data owned by other principals. Information that is owned by a particular user can only be declassified by code that runs with that user's authority. Note that Rule 2 reflects the transitivity of the acts-for relation. For example, if a process can act for a principal P, it can act for any principal Q that P can act for, and therefore can declassify data owned by Q.

Analysis of the safety of a piece of code reduces to analysis of the uses of these rules. As we will see, the rules can be largely checked at compile time, assuming the programmer is willing to supply some annotations. Code that performs a relabeling according to Rule 1 can be checked entirely at compile time, but code that performs a relabeling according to Rule 2 requires additional checking, to ensure that the process acts for the owner. In the language discussed later in the paper, we require that the use of Rule 2 be indicated explicitly, since legal but unintended information leaks could occur otherwise.

An important property of these rules is that the join operator does not interfere with relabeling. It is possible to independently relabel a component of a join: 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 Rule 1 because the set of labels forms a lattice. The property also holds for Rule 2 because the join operator ensures that the flow policies of L_3 are not violated.

This property is important because it permits code that is generic with respect to a label (or part of a label) to perform declassification. It is also helpful for writing code like the statistical analysis package in the medical study example. Because Rule 2 can be applied to part of a join, the analysis package can compute answers using values from its private database, then remove its label from the result that it generates. Without this property, it would have to declassify the private database information immediately—forcing it to give up some protection against leaks.

3.5 Channels

Data enters the system through input channels and leaves it through output channels. In either case, it is important that the channel's label reflect reality. For example, if a printer can be read by a number of people, it is important that the output channel to that printer identify all of them, since otherwise an information leak is possible.

Therefore, channel creation is a sensitive operation since it must get the labels right. We assume it is done by trusted code, which makes use of its understanding of the physical devices to determine whether the requested channel should be created, i.e., whether the label proposed by the program attempting to create the channel matches reality. The channel creator might additionally rely on authentication, e.g., when a user logs on, the authentication or login process might associate the user's principal with the channel for the display being used.

A value read from an input channel is labeled with the channel's label. This is the same as what happens when reading a value from a slot. However, the rule for writing to an output channel is different from writing to a slot, since owners don't matter for writing. Instead, the rule is the following: for any reader R of the output channel, the effective readers of the value's label must include a reader R' such that R can act for R'. This rule ensures that the data can read only by readers who have been authorized to see it. The reason to use the acts-for relation is that by authorizing some R', we have implicitly authorized all principals who can act for R'. This flexibility provides functionality not easily attainable through other mechanisms such as declassification. For example, it allows us to write data that is readable by a group principal to a channel that is readable by a member of the group, since the member principal can act for the group principal.

3.6 Security Class Lattice

If we consider just Rule 1, the set of labels forms a conventional security-class lattice [Den76], where each element in the lattice is one of the possible labels. Labels exist in a partial order as defined by the restriction operator, $\sqsubseteq$. The least restrictive label, denoted by $\perp$, corresponds to data that can flow anywhere; the greatest 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 can always be relabeled to flow upward in the lattice, because restriction does not create a possible information leak.

The lattice has a well-defined *meet* operator, $\sqcap$. Its definition is precisely dual to that of $\sqcup$: it takes the intersection of the owners and the unions of the readers. The meet operator yields the most restrictive label that is strictly less restrictive than its operands. This operator doesn't seem to be very useful in describing computation, and its use is avoided in order to preserve the ability to easily infer labels, as shown in Section 6.

The effect of declassification is that each principal has access to certain relabelings that do not accord with the lattice. However, these relabelings do not leak information.

4 Checking Labels

Labels can be used to annotate code, and the annotated code can be checked statically to verify that it contains no information leaks. In this section, we discuss some issues related to static analysis of annotated code, though we defer issues of how to extend a programming language till the next section. In Section 4.1, we explain the importance of static checking to information flow control, and the problem of *implicit flows* [DD77]. In Section 4.2, we describe a simple way to prevent implicit flows from leaking information by using static analysis.

4.1 Static vs. Dynamic Checking

Information flow checks can be viewed as 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. 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. Access control checks are usually performed at run time, although some access control checks may be performed at compile time [JL78, RSC92]. In general, it seems that some access control checks must be performed dynamically in order to give the system sufficient flexibility.

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 exactly the fact that they can *fail*. In 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 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.

```
x := 0
if b then
   x := 1
end
```

Figure 3: Implicit information flow

Implicit information flows [DD77] are difficult to prevent without static analysis. For example, consider the segment of code shown in Figure 3 and assume that the storage locations b and x belong to different security classes $\underline{b}$ and $\underline{x}$, respectively. (We will follow the literature in using the notation $\underline{e}$ to refer to the label of the expression e.) In particular, assume b is more sensitive than x (more generally, $\underline{b} \not\sqsubseteq \underline{x}$), 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 contains the value of b. A run-time check can easily detect 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. The fact of the program's aborting or continuing implicitly communicates information about the value of b, which can be used in at least 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 evaluating 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.

To provide flexibility, some information flow checks are desirable at run time; such checks are allowed as long as their success or failure does not communicate information improperly—which must itself be checked statically! We examine run-time information flow checks later, in Section 5.10.

4.2 Basic Block Labels

As described in Section 3.1, when a data value is extracted from a slot, it acquires the slot label. Furthermore, to ensure that writing to a slot does not leak information, the label on the slot must be more restrictive than the label on the data

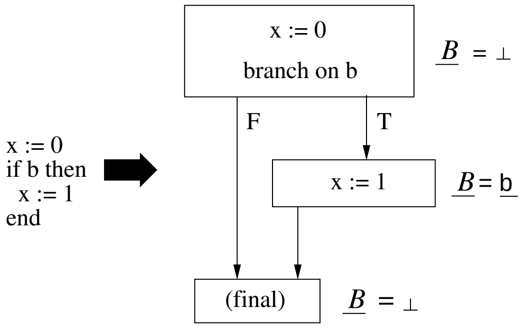

Figure 4: Basic blocks for an if statement

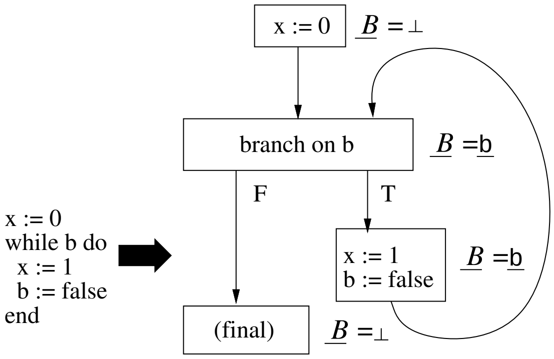

Figure 5: Basic blocks for a while statement

value being written: x := v is legal only if $\underline{v} \sqsubseteq \underline{x}$. However, while this restriction condition is *necessary*, it is not *sufficient* for avoiding information leaks, because of the possibility of implicit information flows. The code in Figure 3 provides an example. The variables x and b are both slots. The expressions 0 and 1 do not give any information about other data, so they are labeled by $\perp$. Therefore the assignment x := 1 appears to be legal. However, earlier we showed that this assignment may be an information leak. Therefore, our simple rule is not sufficient. The problem becomes clearer if we rewrite the if statement:

x := (if b then 1 else x)

Clearly, the value of x after the statement completes is dependent on the value of b, and the assignment is legal only if $\underline{b} \sqsubseteq \underline{x}$.

In general, the flow of control within a program depends on the values of certain expressions. At any given point during execution, various values v_i have been observed in order to decide to arrive at the current program location. Any mutation that occurs can potentially leak information about the observed values v_i, so the slot that is being mutated must be at least as restricted as the labels on all these variables:

$$\bigsqcup_i \underline{v_i} = \underline{v_1} \sqcup \underline{v_2} \sqcup \ldots \sqcup \underline{v_n}$$

This label $\bigsqcup_i \underline{v_i}$ can be determined through straightforward static analysis of the program's basic block diagram, and will be called the *basic block label*, $\underline{B}$. The basic block label indicates information that might be inferred by knowing that the program is executing this basic block. Using the basic block label, we can write the correct rule for checking assignment: assignment to a variable x with a value v is permitted only if $\underline{v} \sqcup \underline{B} \sqsubseteq \underline{x}$.

Intuitively, a basic block label must include the labels of all values that were observed to reach that point in the execution. For example, consider the basic block diagram shown in Figure 4, which corresponds to the code of Figure 3; each basic block, represented as a box in the diagram, is characterized by a single basic block label, and has one or two exit points. Here the basic block for x := 1 has label $\underline{b}$ because the value of b had to be observed to reach that point in the program. However, the label of the "final" block is $\perp$ because at that point the program has no knowledge of any values. It is true that the program could discover information by performing tests that read from slots (e.g., x); however, the basic block label captures what the program knows without any further reading.

Labels of basic blocks are derived as follows. The decision about which exit point to follow from a block B_i is made based on the observation of some value v_i. The label $\underline{B}$ for a particular basic block B is the join of some of the labels $\underline{v_i}$. A label $\underline{v_i}$ is included in the join if it is possible to reach B from B_i, and it is also possible to reach the final node from B_i without passing through B (if all paths from B_i to the final node pass through B, then arriving at B conveys no information about v_i.) The set of $\underline{v_i}$ that must be included in each $\underline{B}$ can be efficiently computed using standard compiler techniques. This rule for basic block label propagation is equivalent to the rule of Denning and Denning [DD77].

Now, consider the execution of a "while" statement, which creates a loop in the basic block diagram. This situation is illustrated in Figure 5. Note that for the final basic block, we obtain $\underline{B} = \perp$ by reasoning in the same way as we did for the "if" statement. This labeling might seem strange, since when we arrive at the final block, we know the value of b. However, arriving at the final block gives no information about the value of b before the code started, and there is no way to use code of this sort to improperly transmit information.

This labeling rule holds as long as all programs terminate, or at least as long as there is no way to derive information from the non-termination of a program [DD77, AR80]. The way one decides that a program has not terminated is to time its execution, either explicitly or through asynchronous communication with another thread. We do not address timing channels in this paper.

If the language allows the raising of exceptions and of return statements, the returned value must also be labeled by the label of the basic block that contains the return or raise.

This fact can be seen clearly by converting a procedure that uses a return statement into one that uses boolean variables to keep track of control flow.

5 Application to a Language

In this section we define a simple programming language that incorporates our model. The goal of this exposition is not to seriously propose a programming language, but to demonstrate that the information flow model can be applied practically to a rich computational model, providing sound and acceptably precise constraints on information flow. These annotations could be applied to other programming models, such as compiled code or JVM code [LY96].

The language supports the usual simple types: integers, strings, records, and arrays of any legal type including other array types. Procedures may contain variable declarations, assignments, if statements, and while statements; they return results by assigning to special return variables, as in Pascal. Variables of record or array types are references to storage on the heap, as in Java [GJS96] and CLU [LAB+84], so that assignment of a record or array (e.g., r1 := r2 or a1 := a2) makes the variables aliases for each other.

For simplicity, the language has no exceptions. Exceptions complicate the propagation of basic block labels, but can be handled using the basic-block propagation described in Section 4.2, assuming that procedures declare the exceptions they might raise. Not having exceptions makes some programs clumsy, but programs written with exceptions can be straightforwardly translated into programs without exceptions, so there is no loss of generality.

For simplicity of presentation, the language also lacks global variables. In our simple language, the first basic block in a procedure has label $\perp$. Global variables could be supported by allowing procedures to accept a special parameter that defines the label of their first basic block.

The language is extended with a few unusual features to support information flow control:

- All variables, arguments, and procedure return values have labeled types. If a type is unlabeled, the label is either a parameter or is inferred, depending on context.
- An explicit "declassify" operator allows the declassification of information.
- A procedure can explicitly test with the if_acts_for statement whether it is able to act for some principal, and if it is, it may use the authority of that principal.
- 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 base type label, which permits run-time label checking. Variables of type label and argument-label parameters may be used to construct labels for types that are mentioned within the procedure body.
- A labelcase 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.

```
pinfo = record [ names, passwords: string{chkr: chkr} ]

check_password (db: array[pinfo{⊥}]{⊥},
                user: string {⊥},
                password: string{client: chkr})
    returns (ret: bool{client: chkr})
    % Return whether password is the password of user

    i: int {chkr: chkr} := 0                  % ⊥
    match: bool {client: chkr;                 %
                 chkr: chkr} := false          % ⊥
    while i < db.length() do                  % ⊥
      if db[i].names = user &                  % ⊥
        db[i].passwords = password then        %
          match := true                        % {client: chkr;
      end                                      %   chkr: chkr}
      i := i + 1                               % ⊥
    end
    ret := false                                  % ⊥
    if_acts_for(check_password, chkr) then        % ⊥
      ret := declassify(match, {client: chkr})    % ⊥
    end
end check_password
```

Figure 6: Annotated password checker

We begin with an example of a program to illustrate some of the features in the language. Then we define the language in more detail, including how to check the constructs using label-checking rules.

5.1 An Example

Figure 6 shows the check_password procedure, which accepts a database of passwords, a password, and a user name, and returns a boolean indicating whether the string is the right password for that user. This example is simple, yet it uses declassification to control information flow in a fine-grained way that is not possible under any previous model of which we are aware.

Two principals are mentioned in this code: chkr represents the password checker role, and client represents the principal of the calling client. The password database is an array of records giving the passwords for users; it should be protected by encapsulation (e.g., check_password should be a method) but this complexity is avoided here. In a real program, client would be a parameter to the routine rather than a single fixed principal; a more general (and more concise) version of check_password is shown later, in Figure 9.

In the figure, all labels are declared explicitly by annotating types with label expressions in braces. The type $T\{L\}$ describes a value of type T that is restricted by the label $\{L\}$. For example, the password argument is readable by the checker but owned by the client, which ensures that the checker can examine the password but cannot distribute it further. The annotations are onerous in this case partly because variable labels are not being inferred; Section 6 shows how to infer labels for this procedure.

The comments (beginning with a "%") on the right-hand side of the example indicate the static value of the basic block label, $\underline{B}$, and are not part of the code.

To perform its work, the procedure uses the db database to find the password of the user. As it looks at the data, its basic block label picks up any dependencies. For example, the if predicate examines password and the fields of db[i]; therefore, its body has the basic block label {client: chkr; chkr: chkr}. This means that the label of match must be at least this restrictive, or the assignment to match in the body of the if statement would leak information (i.e., would result in a compile-time error).

However, the client requires a result with label {client: chkr}. To provide such a result, the checker must explicitly declassify match, removing chkr from the owner set. The declassification can be carried out only if the procedure has the proper authority (i.e., can act for chkr). Therefore, the procedure checks whether it runs with this authority (in the if_acts_for statement); in the then clause, the code runs with the authority of the password checker; otherwise, it does not.

Code receives authority by being granted it in the principal hierarchy; each procedure has its own principal that can participate in acts-for relations. Of course, granting code the right to act for a particular principal can only be done by a process that acts for that principal, so chkr must have explicitly granted check_password the right to act for it. The expectation of the author of the procedure check_password is that it has been given this right. However, the procedure is constructed so that it does not leak information if the right to act for chkr is later revoked.

The need for explicit declassification is not just an artifact of our model. The return value of the procedure is owned by the client, which means that the client has complete control over the value. The procedure's result could conceivably be used to determine the contents of the password database through exhaustive search. In this case, the implementor of the procedure has made a conscious decision that the amount of information leaked by the boolean return value is small enough that declassification is acceptable. Note that the decision about declassification is made locally, by code that acts for the owner of the data; no appeal to an external trusted agent is required.

It is not necessary to run the entire procedure under the chkr authority. Instead this authority is used just where it is needed, for the declassification. Also, note that the procedure only needs to declassify its result. It is not required to declassify everything extracted from the password database, and is therefore protected against accidentally leaking the database contents (e.g., if it called a helping procedure that acted on the database).

5.2 Label-Checking Rules

Now, we explain the constructs of the language and the corresponding label-checking rules. The process of verifying a program according to these rules involves two major steps: first, basic block labels are propagated; then, each individual statement is verified in the context of the basic block that contains it. Verifying a statement requires that we check for the satisfaction of corresponding label constraints, which is discussed in more detail in Section 6.

If the language contained exceptions or gotos, $\underline{B}$ would need to be computed through the basic-block label propagation rule of Section 4.2. For our simple constructs, the rules for propagation of $\underline{B}$ can be stated directly; since control flow is determined only by if and while statements, the basic-block propagation rule takes particularly simple forms. Note that in each statement rule, $\underline{B}$ represents the label for the basic block containing the statement. Many of these rules are similar in intent to those found in Denning and Denning [DD77], though these rules are different in that they are expressed without using the meet operator ($\sqcap$).

5.3 Labeled Types

As described, values and slots have labels that restrict information flow. In statically-typed languages, values and slots also have static types that can be said to restrict flow. These two restrictions can be combined, so that labels are considered to be part of the type. This interpretation allows us to use standard type-checking and notions of subtyping to describe information flows.

Every variable, argument, and return type in the language has a *labeled type*, which consists of a base type such as int, plus a static label. For example, the labeled type int$\{L\}$ represents an integer restricted by L. In general, label expressions, consisting of the join of several labels, may appear within the braces. For example, int$\{\mathsf{chkr} : \mathsf{chkr} \sqcup L_2\}$ is restricted by both $\{\mathsf{chkr} : \mathsf{chkr}\}$ and L_2.

The type and label parts of a labeled type act independently. For any two types S and T where S is a subtype of T ($S \leq T$), and for any two labels L_1 and L_2 where $L_1 \sqsubseteq L_2$, $S\{L_1\} \leq T\{L_2\}$ [VSI96]. This formula also implies that $S\{L_1\} \leq T\{L_1 \sqcup L_3\}$, for any other label L_3.

Parametric types such as arrays and records explicitly mention labels on their type parameters. For example, we can form the type array[int{L}]{L2}, which is an array of labeled integers, where the integers have label L, and the array reference has label L2. In record types, similarly, the individual fields have labeled types.

5.4 Assignment

Given an assignment of the form v := e, where v is a variable with type $T\{\underline{v}\}$ and e is an expression with type $S\{\underline{e}\}$, and $S \leq T$, the assignment is legal if $\underline{e} \sqcup \underline{B} \sqsubseteq \underline{v}$, where $\underline{B}$ is the label for the basic block containing the statement. This condition guarantees both that the information in e is not leaked by placing it in the variable v, and that performing the store operation does not leak information because it happens in this particular basic block.

Record and array assignments are similar to variable assignment, except that they may convey information by virtue of the particular record or array being assigned to. Consider record assignment of the form r.f := e. In this statement, r is a record expression with label $\underline{r}$, f is a field of the record type with declared type $T\{\underline{f}\}$, and e is an expression with type $S\{\underline{e}\}$, $S \leq T$. The assignment is legal if $\underline{e} \sqcup \underline{r} \sqcup \underline{B} \sqsubseteq \underline{f}$. This rule is equivalent to the one in Denning and Denning [DD77]. The rule for assignment to an array element is similar, except that the label on the array index is included on the left-hand side. Because $\underline{r}$ appears on the left-hand side of the rule, fields and array elements become immutable if the variable referring to the record or array becomes more protected than the field or element. For example, a record field with $\underline{r} \not\sqsubseteq \underline{f}$ is immutable, since otherwise information could be leaked by assigning to it.

5.5 if and while

The rules for if and while are similar to each other. Assume that e is a legal boolean expression, and that S is an arbitrary statement. The statement "if e then S end" is legal if S is legal given the basic block label $\underline{B} \sqcup \underline{e}$. The same condition guarantees the legality of "while e then S end". The label $\underline{e}$ does not need to be part of the basic block label after the execution of the if or while statement, because we are not considering covert timing channels or covert channels arising from non-termination, as discussed in Section 4.2.

5.6 Authority

A procedure executes with some authority that has been granted to it. Authority may be granted through the principal hierarchy or because the procedure's caller grants the procedure the right to act for other principals.

At any given point within a program, the compiler understands the code to be running with the ability to act for some set of principals, which we call the *effective authority* of the code at that point. The effective authority can never exceed the true authority at any point during execution.

When a procedure starts running, it has no effective authority. It may increase its effective authority by testing whether it has the authority to act for a principal. If the test succeeds, the effective authority is increased to include that principal. This test is accomplished by using the if_acts_for statement:

if_acts_for(P_1, P_2) **then** S_1 [**else** S_2] **end**

(The brackets around the else clause indicate that it is optional.)

In this statement, P_2 names a principal in the principal hierarchy; P_1 names the current procedure or the special keyword caller. If it names the current procedure, it means the procedure's principal, as discussed in Section 5.1. If it names caller, it denotes the principal(s) that the procedure's caller has granted it the right to act for, as discussed later in Section 5.11.

The effect of if_acts_for is to execute the then block if the specified acts-for relationship exists. If the if_acts_for test fails, the else block, if any, is executed with no additional authority. If the test succeeds, the effective authority of the then block is increased to include P_2.

5.6.1 Revocation

It is possible that while a procedure is executing the then part of an if_acts_for statement, the principal hierarchy may change in a way that would cause the test in the statement to fail. In this case, it may be desirable to revoke the code's permission to run with that authority, and we assume the underlying system can do this, by halting the code's process, at some point after the hierarchy changes.

If a running program is killed by a revocation, information may be leaked about what part of the program was being executed. This constitutes a timing channel, and one that can be made slow enough that it is impractical to use.

5.7 Declassification

A program can explicitly declassify a value. The operation

$$\mathsf{declassify}(e, L)$$

relabels the result of an expression e with the label L, using relabeling Rules 1 and 2 as needed.

Declassification is checked statically, using the effective authority at the point of declassification; the authorization for declassification must derive from a containing if_acts_for control structure. Declassification is legal as long as $\underline{e}$ permits declassification to L, which implies the following rule. Let L_A be a label in which every principal in the effective authority is an owner, but with an empty reader set. The most restrictive label that e could have and still be declassifiable to L is $L \sqcup L_A$, so the declassify() expression is legal if $\underline{e} \sqsubseteq L \sqcup L_A$. For example, if the principal A is part of the effective authority, the label $\{A\colon B,C;\ D\colon E\}$ can be declassified to $\{A\colon C;\ D\colon E\}$, since $\{A\colon C;\ D\colon E\} \sqcup \{A\colon \emptyset\} = \{A\colon \emptyset;\ D\colon E\}$, which is more restrictive than $\{A\colon B,C;\ D\colon E\}$.

5.8 Label Polymorphism

Consider a library routine such as the cosine function (cos). It would be infeasible to declare separate versions of cos for every label in the system. Therefore, we allow procedures to be generic with respect to the labels on their arguments, which means that only one cos function need exist.

If a label is omitted on the type of a procedure argument a, the argument label becomes an implicit parameter to the procedure, and may be referred to as $\underline{\text{a}}$ elsewhere in the procedure signature and body. For example, the cosine function is declared as follows:

cos(x: float) **returns** (y: float$\{\underline{\text{x}}\}$)

cos is generic with respect to the label on the argument x, and $\underline{\text{x}}$ is an implicit argument to the routine.

This signature allows cos to be used on any argument, and the label on the return value is always the same as the label on the argument. Since the code of cos does not depend on what $\underline{\text{x}}$ really is, its code need not access the label, so there is no need either to recompile the code for each distinct label or to pass the label at runtime. Therefore, implicit label polymorphism has no run-time overhead.

5.9 Run-time Labels

Implicit labels allow code to be written that is generic with respect to labels on arguments. However, sometimes more power is needed: for example, to model the accounts database of the bank example, where every customer account has a distinct label. To allow such code to be written, we support run-time labels.

A variable of type label may be used both as a first-class value and as a label for other values. For example, procedures can accept arguments with unknown labels, as in the following procedure declaration:

compute(x: int, lb: label) returns (float$\{\underline{\text{x}} \sqcup \text{lb}\}$)

To simplify static analysis, first-class label variables are immutable after initialization. When a label variable is used as a label, it represents an unknown but fixed label. Because labels form a lattice and obey the simple rules of a lattice, static reasoning about this label is straightforward. For example, if the procedure compute contains the assignment z := x + y, where y has type int{lb}, the assignment is valid as long as it can be statically determined that $\underline{\text{z}} \sqsubseteq \text{lb} \sqcup \underline{\text{x}}$. This condition can be checked even when z and y do not declare their labels explicitly, as discussed in Section 6.

Since label is a type, it too must be labeled wherever it is used. Constant labels and implicit label parameters have type label$\{\bot\}$. Declarations of run-time labels can indicate the label's label explicitly; if the label's label is omitted, it treated like any other type: it is inferred in the case of a local variable, and it is implicit in the case of an argument (this is the situation for the lb argument of compute).

In principle, code that is written in terms of implicit labels can be expresssed in terms of run-time labels. We provide implicit labels because when they provide adequate power, they are easier and cheaper to use than run-time labels. For example, without implicit labels the signature of the cos function would be the following:

cos (x: float{lx}, lx: label$\{\bot\}$)
  returns (y: float{lx})

5.10 Labelcase

The labelcase statement is used to determine the run-time label on a value. It effectively allows a program to examine variables of type label. For example, compute might use labelcase to match the label lx against other labels it has available. A labelcase statement has the following form:

labelcase e **as** v
  when L_1 **do** S_1
  when L_2 **do** S_2
  ...
  [**else** S_3]
end

The effect of this statement is to execute the first statement S_i such that $\underline{\text{e}} \sqsubseteq L_i$, introducing a new variable v containing the same value as e, but with the label L_i.

The block label in the arm of the labelcase does not reflect the label of e, but it does reflect the label of e's label. Suppose the type of e is $T\{L_e\}$ where L_e has the type label$\{\underline{L_e}\}$. Similarly, suppose the labels L_i have respective types label$\{\underline{L_i}\}$. The labelcase statement is valid only if each of the statements S_i is valid given a basic block label $\underline{B}_i$:

$$\underline{B}_i = \underline{B} \sqcup \underline{L}_e \sqcup \left(\bigsqcup_{j=1}^{i} \underline{L}_j \right)$$

This formula says that selecting which statement to execute reveals information *about* the labels, but does not reveal anything about information protected *by* the labels. Therefore, the basic block labels $\underline{B}_i$ depend on $\underline{L}_j$ but not on L_j. The reason for joining all the $\underline{L}_j$ up to i is that the arms of the labelcase are checked sequentially, so each arm that fails conveys some information to its successors.

In a labelcase, neither the tested expression e nor the labels L_i on any of the arms may mention the implicit labels on procedure arguments; rather, the labelcase is limited to constant labels and run-time labels. Implicit procedure argument labels are only intended to help write code that doesn't care what the labels are on the arguments. This restriction on the labelcase avoids the need to pass implicit labels to the generic code that uses them, as discussed in Section 5.8. Since labels may be passed as explicit arguments, and values of type label may be used to label types, no power is lost through this restriction.

5.11 Procedures

A procedure definition has the following syntax:

$$\begin{array}{rcl} \textit{procedure} & \rightarrow & \textit{id} \left[\textit{authority} \right] \\ & & (\ \textit{arguments}\ ) \\ & & \left[\textbf{returns}\ (\ \textit{id}\ \textbf{:}\ T_r\ \{\ L_r\ \}\ ) \right] \\ & & \textit{body}\ \textbf{end} \\ \textit{authority} & \rightarrow & \ll \textbf{caller} \gg \\ \textit{arguments} & \rightarrow & a_1 : T_1 \left[\{L_1\}\right], \ldots, a_n : T_n \left[\{L_n\}\right] \end{array}$$

The optional *authority* clause indicates whether the procedure may be granted some authority by its caller. A procedure with an *authority* clause may try to claim this authority by using caller in an if_acts_for statement. The a_i are the names of the arguments to the procedure. The arguments have types T_i and optional labels L_i. As in variable declarations, labels in the arguments and results of a procedure signature may be simple label expressions, including joins of other labels.

A call to a procedure is legal only if each actual argument to the procedure is assignable to the corresponding formal argument. This means that the formals must have labels that are more restrictive than the block label of the caller at the time of the call, i.e., the normal rule for assignment given in Section 5.4 applies here. Additionally, bindings must be found for all the implicit label parameters such that for each explicit or implicit formal argument label L_i and actual argument label L_{ai}, the constraint $L_{ai} \sqcup \underline{B} \sqsubseteq L_i$ holds, where $\underline{B}$ denotes the basic block label of the call site. Determining the bindings for the implicit labels that will satisfy all these constraints is not trivial, since the formal argument labels may be *join* expressions that mention other implicit argument labels, as in the signature f(a: int, b:int {$\underline{\text{a}} \sqcup$ X}), where X is some other label. The efficient solution of such constraint systems is considered in Section 6.

Furthermore, if the authority clause is present, the caller may provide one or more principals that it acts for. Such a call can occur only within the scope of an if_acts_for statement, since otherwise the effective authority of the caller is nil. For example, the following call from the procedure p grants the authority of the principal my_principal to the procedure doit:

```
if_acts_for(p, my_principal) then
  doit≪my_principal≫(...)
end
```

This model for granting authority protects the caller of a procedure because it can select what part of its authority to grant to the procedure. The implementor of the called procedure is also protected, because the procedure uses only the authority its implementor wishes to claim, and only where needed. (This is similar to the CACL model [RSC92], but provides more protection for the implementor.)

```
protect(T: type, lb: label, v: T{lb})
  returns (p: protected[T]{lb})
    % Create a new protected[T] containing
    % the value v and label lb.

get_label(p: protected[T]) returns (lb: label{p})
    % Return the label that is contained in p

get (p: protected[T], expect: label)
      returns (success: bool{p ⊔ expect},
               v: T{expect ⊔ expect ⊔ p})
    % Return the value that is contained in p if
    % the label expect matches the contained label.
    % Set success accordingly.
```

Figure 7: The operations of protected[T]

5.12 protected[T]

Run-time label checking is conveniently accomplished by using the special type protected[T]. A protected[T] is an immutable object that contains two things: a value of type T, and a label that protects the value.

The type protected[T] is particularly useful for implementing structures like file systems, where the information flow policies of individual objects cannot be deduced statically. For example, a file system directory is essentially an array of protected file objects. Navigating the file system requires a series of run-time label checks, but this work is unavoidable.

protected[T] has two methods: get, which extracts the contained value, and get_label, which extracts the contained label. It also has a constructor, protect, which creates a new protected value. The signatures of these operations are shown in Figure 7. The get method requires an *expected label* as an argument. The value is returned only if the expected label is at least as restrictive as the contained label, which is determined at run time. The expected label must be either a constant label or a variable label; implicit labels are not allowed.

If the language is extended to support some form of data abstraction with encapsulation, the type protected can be implemented in a natural manner by using the labelcase statement. Without these extensions, the closest approximation to protected[T]{$\underline{\text{lb}}$} is the following type:

record[lb: label, x: T{lb}]{$\underline{\text{lb}}$} % *immutable*

Like this type, the type protected[T] has the special property that the label on the reference to a protected[T] (the label $\underline{\text{lb}}$) is assumed to be the label on the contained label, lb. This constraint can be maintained because protected[T] is immutable, and because $L_1 \sqsubseteq L_2$ implies protected[T]$\{L_1\} \leq$ protected[T]$\{L_2\}$.

A protected[T] allows information flow via its contained label: one could pass information by storing different labels in protected[T] objects, and then seeing whether get operations succeed. However, the signature of get prevents an information leak because the success result is labeled both by the label on the label being passed in ($\underline{\text{lb}}$), and by p's label ($\underline{\text{p}}$), which is the same as the label on the contained label. Therefore, this information flow does not create a leak.

5.13 The Bank Example

The bank example from Section 2 is a good example of the need for run-time labels. Each customer account is owned by the individual customer, rather than by the bank, which gives the customer more confidence in the privacy offered by the bank. This example can be conveniently written using protected[T] to protect each customer's account with a label specific to the customer, as shown in Figure 8.

For example, the customer's account can be a simple record type, where the customer's name is protected by (and accessible to) the bank, but balance is protected by both the bank label and a customer label that is stored inside the protected[float]. This design gives the customers protection against dissemination of their balance information.

In this code, Bank represents a principal with the ability to declassify the bank_label label. The current balance of an account is obtained by the procedure get_balance, which accepts a first-class label as an argument. If the label customer that is passed to get_balance is at least as restrictive as the label in the account, the balance is returned as a float labeled by customer. The procedure fails either if no customer exists by that name, or if the label passed in is insufficiently protected.

```
account = record [
    name: string{bank_label},
    balance: protected[float]{bank_label} ]

get_balance (name: string, customer: label,
             accts: array[account{⊥}]{⊥})
        returns (success: bool {name ⊔ customer ⊔
                                 customer },
                 balance: float {name ⊔ customer ⊔
                                 customer})
    . . . % find element i containing the right customer
    s: bool{name ⊔ customer ⊔ customer ⊔ bank_label}
    b: bool{name ⊔ customer ⊔ customer ⊔ bank_label}
    s, b := get(accts[i].balance, customer)
    if_acts_for (get_balance, Bank) then
        success := declassify(s, {name ⊔ customer ⊔
                                 customer})
        balance := declassify(b, {name ⊔ customer ⊔
                                 customer})
    end
end get_balance
```

Figure 8: Bank Example

5.14 Output Channels

Output channels show up as the special opaque type "channel" in the language. The type channel denotes an output channel with a hidden reader set, whose members denote the principals who are reading from the channel. Information can be written to a channel using the "write" procedure:

```
write(c: channel{lb}, s: string{lb}, lb: label{lb})
```

When a write is attempted, the label lb is compared against the hidden reader set within the channel. If it passes the tests of the effective reader set that are described in Section 3.5, the write is successful. Otherwise, it silently fails.

It is important that lb capture all the possible information flows that the write will cause, since otherwise write would not perform a sufficiently stringent test against the channel's reader set. Because lb is used to label s, the channel cannot leak information through the contents of the data that is sent out. Because lb is used to label the argument c as well, the channel cannot be used to leak information by choosing among a set of channels to write to. Finally, because lb labels itself, the channel cannot be used to leak information by changing the label that is passed in as the lb argument, and transmitting information by the fact of the write's success or failure.

6 Verification and Label Inference

This section describes code verification and label inference in more detail. It demonstrates that basic block labels, labels on local variables, and labels in declassify expressions can be inferred efficiently and automatically, making it much more convenient to write properly annotated code. We present a small example of this process.

6.1 Label Constraint Systems

As described earlier, procedure verification is a two-step process that first determines basic block labels by propagating the labels of branch-determining expressions, and then checking all the statements in the program in the context of their basic blocks. While the expressions that control branch decisions can be identified statically, their labels depend on the label of the basic block in which they occur. We seem to have a chicken-and-egg problem.

The difficulty can be resolved by creating a label constraint system and solving for all the inferred labels simultaneously. We start by inventing fresh labels L_i for each of the branch expressions in blocks B_i. These labels are propagated according to the rules for determining basic block labels, and fed into the statement verification stage. Verifying each statement requires checking of corresponding constraints involving labels. To verify code, we collect all the constraints that are demanded by the various statements, and solve them as a system. If the constraints are inconsistent and generate a contradiction, the program contains an information leak.

Solving the constraint system is tractable because the constraints all take a particularly simple form. Each constraint has either a label or a join of labels on both the left- and right-hand sides, e.g., $L_1 \sqcup L_2 \sqsubseteq L_3 \sqcup L_4$. This equation is equivalent to the two equations $L_1 \sqsubseteq L_3 \sqcup L_4$ and $L_2 \sqsubseteq L_3 \sqcup L_4$, and in general, we can convert the constraints to a canonical form in which there is only one label (variable or constant) on the left-hand side.

Explicitly declared labels (whether constants or first-class labels) and implicit argument labels are treated as constants; basic block labels and undeclared labels on local variables are treated as variables. The goal of verification is to determine whether there are assignments to the local variable labels and basic block labels that satisfy all the constraints. This problem is similar in form to the problem of satisfying propositional Horn clauses; in fact, a linear-time algorithm for satisfying Horn clauses [DG84, RM96] can be adapted easily to this problem. If, on the other hand, we had permitted use of both the $\sqcup$ and $\sqcap$ operators in constructing label expressions, the label satisfaction problem would have become NP-complete [RM96].

The algorithm works by keeping track of conservative upper bounds for each unknown label. Initially, all the upper bounds are set to $\top$. The algorithm then iteratively decreases the upper bounds, until either all equations are satisfied or a contradiction is observed. At each step, the algorithm picks an equation that is not satisfied when all variables are substituted by their upper bounds. If the unsatisfied equation has a constant label on its left-hand side, a contradiction has been detected. Otherwise, the upper bound estimate for the variable label on the left-hand side is adjusted to be the *meet* ($\sqcap$) of its current upper bound and the value of the right-hand side. In evaluating the right-hand side, all variables are replaced with their current upper bound estimates.

```
check_password(db: array[pinfo{⊥}], user: string,
               pwd: string)
    returns (ret: bool{user ⊔ pwd ⊔ db})
  % Return whether pwd is the password of user

  i: int := 0                              % ⊥
  match: bool := false                     % ⊥
  while i < db.length() do                 % L1
    if db[i].names = user &                % L1
       db[i].passwords = pwd then          %
         match := true                     % L1⊔L2
    end                                    %
    i := i + 1                             % L1
  end                                      %
  ret := false                             % ⊥
  if_acts_for(check_password, chkr)        %
    ret := declassify(match)               % ⊥
  end
end check_password
```

Figure 9: Password example with implicit labels

Like the algorithm for satisfying Horn clauses, this algorithm requires a number of iterations that is linear in the total size of the constraints; the total size of the constraints is at worst quadratic in the length of the code. Therefore, this inference algorithm seems very practical.

The labels found by this algorithm are the most restrictive labels that satisfy the constraints. However, the actual values that the inference algorithm finds are irrelevant, because they are never converted to first-class values of type label. What is important is that there *is* a satisfying assignment to all the labels, proving that the code is safe.

6.2 Inference Example

Figure 9 shows the code for a more flexible version of the check_password procedure that was presented earlier. This version of check_password is usable by any client principal. Because the arguments db, user, and pwd have no declared labels, their labels are implicit parameters to the routine. Note that the local variables i and result do not explicitly declare their labels. The resulting procedure is as safe as the previous version of check_password, and easier to implement and use. Let us now walk through the verification process for this code.

The first step in verification is to construct the basic-block diagram and propagate fresh labels that represent branch

i := 0	$\bot$	$\sqsubseteq$	$\underline{\mathsf{i}}$
match := F	$\bot$	$\sqsubseteq$	$\underline{\mathsf{match}}$
while	$\underline{\mathsf{i}} \sqcup \underline{\mathsf{db}}$	$=$	L_1
if	$\underline{\mathsf{i}} \sqcup \underline{\mathsf{user}} \sqcup \underline{\mathsf{pwd}} \sqcup \underline{\mathsf{db}}$ $\sqcup \{\mathsf{chkr{:}\ chkr}\} \sqcup L_1$	$=$	L_2
match := T	$L_1 \sqcup L_2$	$\sqsubseteq$	$\underline{\mathsf{match}}$
i := i + 1	$\underline{\mathsf{i}} \sqcup L_1$	$\sqsubseteq$	$\underline{\mathsf{i}}$
ret := F	$\bot$	$\sqsubseteq$	$\underline{\mathsf{user}} \sqcup \underline{\mathsf{pwd}} \sqcup \underline{\mathsf{db}}$
declassify	$\underline{\mathsf{match}}$	$\sqsubseteq$	$L_d \sqcup \{\mathsf{chkr} : \emptyset\}$
ret := . . .	L_d	$\sqsubseteq$	$\underline{\mathsf{user}} \sqcup \underline{\mathsf{pwd}} \sqcup \underline{\mathsf{db}}$

Figure 10: Constraints for the password example

$$\begin{aligned} \underline{\mathsf{i}}, \underline{\mathsf{match}}, L_1, L_2 &= \underline{\mathsf{user}} \sqcup \underline{\mathsf{pwd}} \sqcup \underline{\mathsf{db}} \sqcup \{\mathsf{chkr} : \emptyset\} \\ L_d &= \underline{\mathsf{user}} \sqcup \underline{\mathsf{pwd}} \sqcup \underline{\mathsf{db}} \end{aligned}$$

Figure 11: Constraint solutions

expressions. The comments in Figure 9 show the value of the basic block labels for each basic block, in terms of the two branch-expression labels L_1 and L_2 (for the if and while, respectively.)

Next, the code is analyzed to generate the set of label constraints shown in Figure 10, which include inequalities corresponding to the statements in the program, plus some equalities that bind the basic-block branch-expression labels to the labels for the corresponding expressions. The equalities can be transformed into a pair of constraints to preserve the canonical constraint form. Note that the use of declassify generates an additional constraint, introducing a new variable L_d that represents the label of the declassified result. This procedure provides a good example of why it is important for declassification to be able to apply to just one component of a *join*, as discussed in Section 3.4. The fact that declassification works at all in this procedure, let alone is possible to verify automatically, is due to this property of the declassification rule.

Applying the constraint-solving algorithm just described, a single backward pass through the canonical forms of these constraints yields labels that satisfy them, as shown in Figure 11.

7 Related Work

There has been much work on information flow control and on the static analysis of security guarantees. The lattice model of information flow comes from the early work of Bell and LaPadula[BL75] and Denning [Den76]. More recent work on information flow policies has examined complex aggregation policies for commercial applications [CW87, BN89, Fol91]. We have not addressed policies that capture conflicts of interest, though our fine-grained tracking of ownership information seems applicable. Many of these information control models use dynamic labels rather than static labels and therefore cannot be checked statically. IX [MR92] is a good example of a practical information flow control system that takes this approach. Our propagation of ownership information is also reminiscent of models of access control that merge ACLs at run time [MMN90].

Static analysis of security guarantees also has a long history. It has been applied to information flow [DD77, AR80] and to access control [JL78, RSC92]. There has recently been more interest in provably-secure programming languages, treating information flow checks in the domain of type checking [VSI96, Vol97]. Also, integrity constraints [Bib77] have been treated as type checking [PO95].

We have avoided considering covert channels arising from time measurement and thread communication. A scheme for statically analyzing thread communication has been proposed [AR80]; essentially, a second basic block label is added with different propagation rules. This second label is used to restrict communication with other threads. The same technique would remove timing channels, and could be applied to our scheme. It is not clear how well this scheme works in practice; it seems likely to restrict timing and communication quite severely. Static side-effect and region analysis [JG91], which aims to infer all possible side-effects caused by a piece of code, may be able to capture effects like timing channels.

8 Conclusions

This work was motivated by a desire to provide better security when using downloaded applications, by making fine-grained information flow control practical and efficient. This paper is a first step towards this goal.

A key limitation of most multilevel security models is that there is a single, centralized policy on how information can flow. This kind of policy does not match well with the decentralized model in which each user is independent. Our model provides each user the ability to define information flow policy at the level of individual data items. Each user has the power to declassify his own data, but declassification does not leak other users' data.

An important aspect of our label model is that labels identify the owners, or sources, of the data. Each owner listed in the label maintains its own separate annotation (the reader set) to control where its data may flow. When running on behalf of a particular principal P, a program is able to edit P's portion of the label without violating other principals' policies. Labels form a familiar security class lattice, but each principal has access to some non-lattice relabelings (by declassifying).

We have also shown how the labels we define can be used to annotate a simple programming language, which suggests that other programming languages and intermediate code formats can be similarly annotated. To ensure proper information flows, the labels in the resulting code can be statically checked by the compiler, in a manner similar to type checking. Also, as part of the label checking process, the compiler can construct a trace of its checking process. Some form of this trace can accompany the generated code, and can be used later to verify information flows in the code.

Labels are mostly checked statically, which has benefits in space and time. Compiled code does not need to perform checks, so the code is shorter and faster than with run-time checks. Storage locations that are statically typed require no extra space to store a label. However, we have also defined a mechanism for run-time label checking that allows the flexibility of run-time checks when they are truly needed. This mechanism guarantees that when the run-time checks fail, information is not leaked by their failure. The mechanism of implicit label polymorphism also extends the power of static analysis, by allowing the definition of code that is generic with respect to some or all of the labels on its arguments. We have presented a simple algorithm that, despite our extensions to the label system, is able to efficiently infer labels for basic blocks and local variables. Label inference makes the writing of label-safe code significantly less onerous.

We have provided some simple examples of code and software designs that cannot be adequately expressed using previous models of access control or information flow. These examples demonstrate that the new features of user declassification, label polymorphism, and run-time label checking add new power and expressiveness in capturing security policies.

9 Future Work

There are many directions to explore with this new model. An obvious next step is to implement the model by extending an existing language compiler, and developing working applications for examples like those in Section 2.

It should also be possible to augment the Java Virtual Machine [LY96] with annotations similar to those proposed in Section 5. The bytecode verifier would check both types and labels at the time that code is downloaded into the system.

The computational model described in Section 5 has a reasonable set of data types. However, it ought to support user-defined data abstractions, including both parametric and subtype polymorphism.

Formal proofs of the soundness of the model might add some confidence. In the absence of any use of the declassify operation, labels are located in a simple lattice that applies equally to all users, and previous results for the security of lattice-based information flow models apply to this model as well. However, because declassify is intended to allow information to flow across or down the lattice, standard security policies such as non-interference [GM84] are intentionally inapplicable.

Because integrity [Bib77] constraints have a natural lattice structure, supporting them may be an interesting extension to our label model; the label model can be augmented to allow each owner to establish an independent integrity policy, just as each owner now can establish an independent information flow policy.

We have assumed an entirely trusted execution environment, which means that the model described here does not work well in large, networked systems, where varying levels of trust exist among nodes in the network. Different principals may also place different levels of trust in the various nodes. A simple technique for dealing with distrusted nodes is to transmit opaque receipts or tokens for the data. However, more work is needed to adapt our model to this kind of system. It is also likely that the model of output channels should be extended to differentiate among the different kinds of outputs from the system.

We have not considered the possibility of covert channels that arise from timing channels and from asynchronous communication between threads, which can also be used to create timing channels. The technique of having a *timing label* for each basic block, as in Andrews and Reitman [AR80], may help with this problem, but more investigation is needed.

Acknowledgments

The authors would like to acknowledge the helpful comments of the many people who have read this paper, including Martín Abadi, Atul Adya, Kavita Bala, Phil Bogle, Miguel Castro, Steve Garland, Robert Grimm, Butler Lampson, Roger Needham, Matt Stillerman, and the anonymous reviewers.

References

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

[Bib77] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, April 1977.

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

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

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

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

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

[DG84] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulæ. *Journal of Logic Programming*, 1(3):267–284, October 1984.

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

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, August 1996. ISBN 0-201-63451-1.

[GM84] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 11–20, April 1984.

[JG91] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *ACM Symposium on Principles of Programming Languages*, pages 303–310, January 1991.

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

[LAB+84] Barbara Liskov, Russell Atkinson, Toby Bloom, J. Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*. Springer-Verlag, 1984. Also published as Lecture Notes in Computer Science 114, G. Goos and J. Hartmanis, Eds., Springer-Verlag, 1981.

[LABW91] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. In *Proc. 13th ACM Symp. on Operating System Principles (SOSP)*, pages 165–182, October 1991. *Operating System Review*, 253(5).

[Lam73] Butler W. Lampson. A note on the confinement problem. *Comm. of the ACM*, 10:613–615, 1973.

[LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine*. Addison-Wesley, Englewood Cliffs, NJ, May 1996.

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

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

[Nec97] George C. Necula. Proof-carrying code. In *Proc. of ACM Symp. on Principles of Programming Languages*, pages 106–119, January 1997.

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

[RM96] Jakob Rehof and Torben Æ. Mogensen. Tractable constraints in finite semilattices. In *Proc. 3rd International Symposium on Static Analysis*, number 1145 in Lecture Notes in Computer Science, pages 285–300. Springer-Verlag, September 1996.

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

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

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