Disjunction Category Labels

Deian Stefan[1], Alejandro Russo[2], David Mazières[1], and John C. Mitchell[1]

[1] Stanford University
[2] Chalmers University of Technology

Abstract. We present disjunction category (DC) labels, a new label format for enforcing information flow in the presence of mutually distrusting parties. DC labels can be ordered to form a lattice, based on propositional logic implication and conjunctive normal form. We introduce and prove soundness of decentralized privileges that are used in declassifying data, in addition to providing a notion of privilege-hierarchy. Our model is simpler than previous decentralized information flow control (DIFC) systems and does not rely on a centralized principal hierarchy. Additionally, DC labels can be used to enforce information flow both statically and dynamically. To demonstrate their use, we describe two Haskell implementations, a library used to perform dynamic label checks, compatible with existing DIFC systems, and a prototype library that enforces information flow statically, by leveraging the Haskell type checker.

Keywords: Security, labels, decentralized information flow control, logic

1 Introduction

Information flow control (IFC) is a general method that allows components of a system to be passed sensitive information and restricts its use in each component. Information flow control can be used to achieve confidentiality, by preventing unwanted information leakage, and integrity, by preventing unreliable information from flowing into critical operations. Modern IFC systems typically label data and track labels, while allowing users exercising appropriate privileges to explicitly downgrade information themselves. While the IFC system cannot guarantee that downgrading preserves the desired information flow properties, it is possible to identify all the downgrading operations and limit code audit to these portions of the code. Overall, information flow systems make it possible to build applications that enforce end-to-end security policies even in the presence of untrusted code.

We present disjunction category (DC) labels: a new label format for enforcing information flow in systems with mutually distrusting parties. By formulating DC labels using propositional logic, we make it straightforward to verify conventional lattice conditions and other useful properties. We introduce and prove soundness of decentralized privileges that are used in declassifying data, and provide a notion of privilege-hierarchy. Compared to Myers and Liskov's decentralized label model (DLM) [21], for example, our model is simpler and does not

rely on a centralized principal hierarchy. Additionally, DC labels can be used to enforce information flow both statically and dynamically, as shown in our Haskell implementations.

A DC label, written $\langle S, I \rangle$, consists of two Boolean formulas over principals, the first specifying secrecy requirements and the second specifying integrity requirements. Information flow is restricted according to implication of these formulas in a way that preserves secrecy and integrity. Specifically, secrecy of information labeled $\langle S, I \rangle$ is preserved by requiring that a receiving channel have a stronger secrecy requirement S' that *implies* S, while integrity requires the receiver to have a weaker integrity requirement I' that is *implied by* I. These two requirements are combined to form a can-flow-to relation, which provides a partial order on the set of DC labels that also has the lattice operations meet and join.

Our decentralized privileges can be delegated in a way that we prove preserves confidentiality and integrity properties, resulting in a privilege hierarchy. Unlike [21], this is accomplished without a notion of "can act for" or a central principal hierarchy. Although our model can be extended to support revocation using approaches associated with public key infrastructures, we present a potentially more appealing selective revocation approach that is similar to those used in capability-based systems.

We illustrate the expressiveness of DC labels by showing how to express several common design patterns. These patterns are based in part on security patterns used in capability-based systems. Confinement is achieved by labeling data so that it cannot be read and exfiltrated to the network by arbitrary principals. A more subtle pattern that relies on the notion of clearance is used to show how a process can be restricted from even accessing overly-sensitive information (e.g., private keys); this pattern is especially useful when covert channels are a concern. We also describe privilege separation and user authentication patterns. As described more fully later in the paper, privilege separation may be achieved using delegation to subdivide the privileges of a program and compartmentalize a program into components running with fewer privileges. The user authentication pattern shows how to leverage a login client that users trust with their username and password (since the user supplies them as input), without unnecessarily creating other risks.

We describe two Haskell implementations: a library used to perform dynamic label checks, compatible with existing DIFC systems, and a prototype library that enforces information flow statically by leveraging Haskell's type checker.

The remainder of the paper is structured as follows. In Section 2, we introduce DC labels and present some of their properties. Section 3 presents semantics and soundness proofs for our DC label system. Design patterns are presented and explained in Section 5, with the implementations presented in Section 6. We summarize related work in Section 7 and conclude in Section 8.

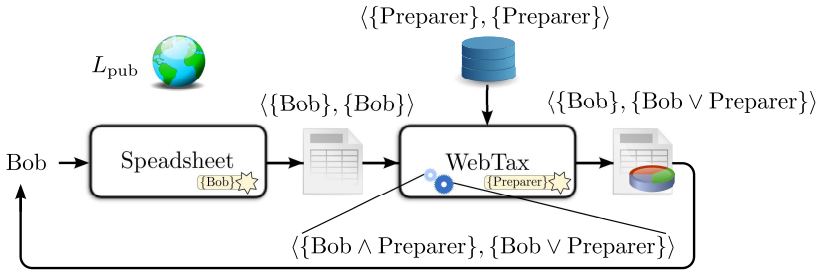

Fig. 1: A tax preparation system with mutually distrusting parties.

2 DC Label Model

In a DIFC system, every piece of data is *labeled*, or "tagged." Labels provide a means for tracking, and, more importantly, controlling the propagation of information according to a security policy, such as *non-interference* [10].

DC labels can be used to express a conjunction of restrictions on information flow that represents the interests of multiple stake-holders. As a result, DC labels are especially suitable for systems in which participating parties do not fully trust each other. Fig. 1 presents an example, originally given in [21], that illustrates such a system. Here, user Bob firstly inputs his tax information into the *Spreadsheet* program, which he fully trusts. The data is then exported to another program, called *WebTax*, for final analysis. Though conceptually simple, several challenges arise since Bob does not trust WebTax with his data. Without inspecting WebTax, Bob cannot be sure that his privacy policies are respected and his tax information is not exfiltrated to the network. Analogously, the WebTax author, called Preparer, does not entrust Bob with the source code. Furthermore, the tax preparation program relies on a proprietary database and Preparer wishes to assert that even if the program contains bugs, the proprietary database information cannot be leaked to the public network. It is clear that even for such a simple example the end-to-end guarantees are difficult to satisfy with more-traditional access control mechanisms. Using IFC, however, these security policies can be expressed naturally and with minimal trust. Specifically, the parties only need to trust the system IFC-enforcement mechanism; programs, including WebTax, can be executed with no implicit trust. We now specify DC labels and show their use in enforcing the policies of this example.

As previously mentioned, a DC label consists of two Boolean formulas over principals. We make a few restrictions on the labels' format in order to obtain a unique representation for each label and an efficient and decidable can-flow-to relationship.

Definition 1 (DC Labels). *A* DC label*, written $\langle S, I \rangle$, is a pair of Boolean formulas over principals such that:*

- *Both S and I are minimal formulas in conjunctive normal form (CNF), with terms and clauses sorted to give each formula a unique representation, and*

– *Neither S nor I contains any negated terms.*

In a DC label, S protects secrecy by specifying the principals that are allowed (or whose consent is needed) to observe the data. Dually, I protects integrity by specifying principals who created and may currently modify the data. For example, in the system of Fig. 1, Bob and Preparer respectively label their data $\langle \{\text{Bob}\}, \{\text{Bob}\} \rangle$ and $\langle \{\text{Preparer}\}, \{\text{Preparer}\} \rangle$, specifying that they created the data and they are the only observers.

Data may flow between differently labeled entities, but only in such a way as to accumulate additional secrecy restrictions or be stripped in integrity ones, not vice versa. Specifically there is a partial order, written $\sqsubseteq$ ("can-flow-to"), that specifies when data can flow between labeled entities. We define $\sqsubseteq$ based on logical implication ($\implies$) as follows:

Definition 2 (can-flow-to relation). *Given any two DC labels $L_1 = \langle S_1, I_1 \rangle$ and $L_2 = \langle S_2, I_2 \rangle$, the can-flow-to relation is defined as:*

$$\frac{S_2 \implies S_1 \qquad I_1 \implies I_2}{\langle S_1, I_1 \rangle \sqsubseteq \langle S_2, I_2 \rangle}$$

In other words, data labeled $\langle S_1, I_1 \rangle$ can flow into an entity labeled $\langle S_2, I_2 \rangle$ as long as the secrecy of the data, and integrity of the entity are preserved. Intuitively, the $\sqsubseteq$ relation imposes the restriction that any set of principals who can observe data afterwards must also have been able to observe it earlier. For instance, it is permissible to have $S_2 = \{\text{Bob} \wedge \text{Preparer}\}$ and $S_1 = \text{Bob}$, because $S_2 \implies S_1$, and Bob's consent is still required to observe data with the new label. Dually, integrity of the entity is preserved by requiring that the source label impose more restrictions than that of the destination.

In our model, public entities (e.g., network interface in Fig. 1) have the default, or *empty* label, $\langle \mathbf{True}, \mathbf{True} \rangle$, written L_{pub}. Although specified by the label $\langle S, I \rangle$, it is intuitive that data labeled as such can be written to a public network with label L_{pub}, only with the permission of a set of principals satisfying the Boolean formula S. Conversely, data read from the network can be labeled $\langle S, I \rangle$ only with the permission of a set of principals satisfying I.

In an IFC system, label checks using the can-flow-to relation are performed at every point of possible information flow. Thus, if the WebTax program of Fig. 1 attempts to write Bob or Preparer's data to the network interface, either by error or malfeasance, both label checks $\langle \{\text{Bob}\}, \{\text{Bob}\} \rangle \sqsubseteq L_{\text{pub}}$ and $\langle \{\text{Preparer}\}, \{\text{Preparer}\} \rangle \sqsubseteq L_{\text{pub}}$ will fail. However, the system must also label the intermediate results of a WebTax computation (on Bob and Preparer's joint data) such that they can only be observed and written to the network if both principals consent.

The latter labeling requirement is recurring and directly addressed by a core property of many IFC systems: the label lattice property [4]. Specifically, for any two labels L_1, L_2 the lattice property states that there is a well defined, *least upper bound* (*join*), written $L_1 \sqcup L_2$, and *greatest lower bound* (*meet*), written

$L_1 \sqcap L_2$, such that $L_i \sqsubseteq L_1 \sqcup L_2$ and $L_1 \sqcap L_2 \sqsubseteq L_i$ for L_i and $i = 1, 2$. We define the join and meet for DC labels as follows.

Definition 3 (Join and meet for DC labels). *The join and meet of any two DC labels $L_1 = \langle S_1, I_1 \rangle$ and $L_2 = \langle S_2, I_2 \rangle$ are respectively defined as:*

$$L_1 \sqcup L_2 = \langle S_1 \wedge S_2, I_1 \vee I_2 \rangle$$
$$L_1 \sqcap L_2 = \langle S_1 \vee S_2, I_1 \wedge I_2 \rangle$$

where each component of the resulting labels is reduced to CNF.

Intuitively, the secrecy component of the join protects the secrecy of L_1 and L_2 by specifying that both set of principals, those appearing in S_1 and those in S_2, must consent for data labeled $S_1 \wedge S_2$ to be observed. Conversely, the integrity component of the join, $I_1 \vee I_2$, specifies that either principals of I_1 or I_2 could have created and modify the data. Dual properties hold for the meet $L_1 \sqcap L_2$, a label computation necessary when labeling an object that is written to multiple entities. We note that although we use $I_1 \vee I_2$ informally, by definition, a DC label component must be in CNF. Reducing logic formulas, such as $I_1 \vee I_2$, to CNF is standard [23], and we do not discuss it further.

Revisiting the example of Fig. 1, we highlight that the intermediate results generated by the WebTax program from both Bob and Preparer's data are labeled by the join $\langle \{\text{Bob}\}, \{\text{Bob}\} \rangle \sqcup \langle \{\text{Preparer}\}, \{\text{Preparer}\} \rangle$ which is reduced to $\langle \{\text{Bob} \wedge \text{Preparer}\}, \{\text{Bob} \vee \text{Preparer}\} \rangle$. The secrecy component of the label confirms our intuition that the intermediate results are composed of both party's data and thus the consent of both Bob and Preparer is needed to observe it. In parallel, the integrity component agrees with the intuition that the intermediate results could have been created from Bob or Preparer's data.

2.1 Declassification and endorsement

We model both declassification and endorsement as principals explicitly deciding to exercise *privileges*. When code exercises privileges, it means code acting on behalf of a combination of principals is requesting an action that might violate the can-flow-to relation. For instance, if the secrecy component of a label is $\{\text{Bob} \wedge \text{Preparer}\}$, then by definition code must act on behalf of both Bob and the Preparer to transmit the data over a public network. However, what if the Preparer unilaterally wishes to change the secrecy label on data from $\{\text{Bob} \wedge \text{Preparer}\}$ to $\{\text{Bob}\}$ (as to release the results to Bob)? Intuitively, such a partial declassification should be allowed, because the data still cannot be transmitted over the network without Bob's consent. Hence, if the data is eventually made public, both Bob and the Preparer will have consented, even if not simultaneously.

We formalize such partial declassification by defining a more permissive preorder, $\sqsubseteq_P$ ("can-flow-to given privileges P"). $L_1 \sqsubseteq_P L_2$ means that when exercising privileges P, it is permissible for data to flow from an entity labeled L_1

to one labeled L_2. $L_1 \sqsubseteq L_2$ trivially implies $L_1 \sqsubseteq_P L_2$ for any privileges P, but for non-empty P, there exist labels for which $L_1 \sqsubseteq_P L_2$ even though $L_1 \not\sqsubseteq L_2$.

We represent privileges P as a conjunction of principals for whom code is acting. (Actually, P can be a more general Boolean formula like label components, but the most straight-forward use is as a simple conjunction of principals.) We define $\sqsubseteq_P$ as follows:

Definition 4 (can-flow-to given privileges relation). *Given a Boolean formula P representing privileges and any two DC labels $L_1 = \langle S_1, I_1 \rangle$ and $L_2 = \langle S_2, I_2 \rangle$, the can-flow-to given privileges P relation is defined as:*

$$\frac{P \wedge S_2 \implies S_1 \qquad P \wedge I_1 \implies I_2}{\langle S_1, I_1 \rangle \sqsubseteq_P \langle S_2, I_2 \rangle}$$

Recall that without exercising additional privileges, data labeled $\langle S, I \rangle$ can be written to a public network, labeled L_{pub}, only with the permission of a set of principals satisfying the Boolean formula S, while data read from a public network can be labeled $\langle S, I \rangle$ only with the permission of a set of principals satisfying I. Considering additional privileges, it is easy to see that $\langle S, I \rangle \sqsubseteq_P L_{\text{pub}}$ iff $P \implies S$ and, conversely, $L_{\text{pub}} \sqsubseteq_P \langle S, I \rangle$ iff $P \implies I$. In other words, code exercising privileges P can declassify and write data to the public network if P implies the secrecy label of that data, and can similarly incorporate and endorse data from the public network if P implies the integrity label.

In our WebTax example, the Spreadsheet program runs on behalf of Bob and exercises the {Bob} privilege to endorse data sent to WebTax. Conversely, the WebTax program is executed with the {Preparer} privilege which it exercises when declassifying results from {Bob$\wedge$Preparer} to {Bob}; as expected, to allow Bob to observe the results, this declassification step is crucial.

It is a property of our system that exporting data through multiple exercises of privilege cannot reduce the overall privilege required to export data. For instance, if $\langle S, I \rangle \sqsubseteq_{P_1} \langle S', I' \rangle \sqsubseteq_{P_2} L_{\text{pub}}$, it must be that $P_1 \wedge P_2 \implies S$, since $P_2 \implies S'$ and $P_1 \wedge S' \implies S$. A similar, and dual, property holds for multiple endorsements.

The mechanisms provided by $\sqsubseteq_P$ corresponds to the *who* dimension of declassification [25], i.e., whoever has the privileges P can use the relationship $\sqsubseteq_P$ to release (endorse) information. With minimal encoding, it is also possible to address the *what* and *when* dimension using $\sqsubseteq_P$. Specifically, the what dimension can be addressed by carefully designing the data type in such a way that there is an explicit distinction on what part of the data is allowed to be released. The *when* dimension, on the other hand, consists on designing the trusted modules in such a way that certain privileges can only be exercised when some, well-defined, events occurs.

In our model, privileges can be *delegated*. Specifically, a process may delegate privileges to another process according to the following definition:

Definition 5 (Can-delegate relation). *A process with privilege P can delegate any privilege P', such that $P \implies P'$.*

In other words, it is possible to delegate a privilege P' that is at most as strong as the parent privilege P. In Section 5, we give a concrete example of using delegation to implement a privilege separation.

2.2 Ownership and categories

Our definition of DC label components as conjunctions of clauses, each imposing an information flow restriction, is similar to the DStar [31] label format which uses a set of *categories*, each of which is used to impose a flow restriction. Though the name category may be used interchangeably with clause, our categories differ from those of DStar (or even DLM) in that they are disjunctions of principals—hence the name, *disjunction category* labels.

The principals composing a category are said to *own* the category—every owner is trusted to uphold or bypass the restriction imposed by the category. For instance, the category [Bob $\vee$ Alice] is owned by both Alice and Bob. We can thus interpret the secrecy component $\{[\text{Bob} \vee \text{Alice}] \wedge \text{Preparer}\}$ to specify that data can be observed by the Preparer in collaboration with *either* Bob or Alice. Though implicit in our definition of a DC label, this joint ownership of a category allows for expressing quite complex policies. For example, to file joint taxes with Alice, Bob can simply labels the tax data $\langle\{[\text{Bob} \vee \text{Alice}]\}, \{\text{Bob}\}\rangle$, and now the WebTax results can be observed by both him and Alice. Expressing such policies in other systems, such as DLM or DStar, can only be done through external means (e.g., by creating a new principal AliceBob and encoding its relationship to Alice and Bob in a centralized principal hierarchy).

In the previous section we represent privileges P as a conjunction of principals for whom code is acting. Analogous to a principal owning a category, we say that a process (or computation) *owns* a principal if it acting or running on its behalf. (More generally, the code is said to own all the categories that compose P.)

3 Soundness

In this section, we show that the can-flow-to relation ($\sqsubseteq$) and the relation ($\sqsubseteq_P$) for can-flow-to given privileges P satisfy various properties. We first show that $\sqsubseteq$, given in Definition 2, is partial order.

Lemma 1 (DC labels form a partially ordered set). *The binary relation $\sqsubseteq$ over the set of all DC labels is a partial order.*

Proof. Reflexivity and transitivity follow directly from the Reflexivity and transitivity of ($\implies$). By Definition 1, the components of a label, and thus the label, have a unique representation. Directly, the antisymmetry property holds.

Recall from Section 2 that for any two labels L_1 and L_2 there exists a join $L_1 \sqcup L_2$ and meet $L_1 \sqcap L_2$. The join must be the least upper bound of L_1 and L_2, with $L_1 \sqsubseteq L_1 \sqcup L_2$, and $L_2 \sqsubseteq L_1 \sqcup L_2$; conversely, the meet must be the greatest lower bound of L_1 and L_2, with $L_1 \sqcap L_2 \sqsubseteq L_1$ and $L_1 \sqcap L_2 \sqsubseteq L_2$. We prove these properties and show that DC labels form a lattice.

Proposition 1 (DC labels form a bounded lattice). *DC labels with the partial order relation $\sqsubseteq$, join $\sqcup$, and meet $\sqcap$ form a bounded lattice with minimum element $\bot = \langle \mathbf{True}, \mathbf{False} \rangle$ and maximum element $\top = \langle \mathbf{False}, \mathbf{True} \rangle$.*

Proof. The lattice property follows from Lemma 1, the definition of DC labels, and the definition of the join and meet as given in Definition 3.

It is worth noting that the DC label lattice is actually product lattice, i.e., a lattice where components are elements of a secrecy and (a dual) integrity lattice [29].

In Section 2.1 we detailed declassification and endorsement of data in terms of exercising privileges. Both actions constitute bypassing restrictions of $\sqsubseteq$ by using a more permissive relation $\sqsubseteq_P$. Here, we show that this privilege-exercising relation, as given in Definition 4, is a pre-order and that privilege delegation respects its restrictions.

Proposition 2 (The $\sqsubseteq_P$ relation is a pre-order). *The binary relation $\sqsubseteq_P$ over the set of all DC labels is a pre-order.*

Proof. Reflexivity and transitivity follow directly from the reflexivity and transitivity of ($\implies$). Unlike $\sqsubseteq$, however, $\sqsubseteq_P$ is not necessarily antisymmetric (showing this, for a non-empty P, is trivial).

Informally, exercising privilege P may allow a principal to ignore the distinction between certain pairs of clauses, hence $\sqsubseteq_P$ is generally not a partial order. Moreover, the intuition that $\sqsubseteq_P$, for any non-empty P, is always more permissive than $\sqsubseteq$ follows as a special case of the following proposition.

Proposition 3 (Privileges substitution). *Given privileges P and P', if $P \implies P'$ then P can always be substituted in for P'. Specifically, for all labels L_1 and L_2, if $P \implies P'$ and $L_1 \sqsubseteq_{P'} L_2$ then $L_1 \sqsubseteq_P L_2$.*

Proof. First, we note that if $P \implies P'$, then for any X, X', such that $X \wedge P' \implies X'$, the proposition $X \wedge P \implies X \wedge P' \implies X'$ holds trivially. By Definition 4, $L_1 \sqsubseteq_{P'} L_2$ is equivalent to: $S_2 \wedge P' \implies S_1$ and $I_1 \wedge P' \implies I_2$. However, from $P \implies P'$, we have $S_2 \wedge P \implies S_2 \wedge P' \implies S_1$, and $I_1 \wedge P \implies I_1 \wedge P' \implies I_2$. Correspondingly, we have $L_1 \sqsubseteq_P L_2$.

Informally, if a piece of code exercises privileges P' to read or endorse a piece of data, it can do so with P as well. In other words, $\sqsubseteq_P$ is at least at as permissive as $\sqsubseteq_{P'}$. Letting $P' = \mathbf{True}$, it directly follows that for any non-empty P, i.e., for $P \neq \mathbf{True}$, the relation $\sqsubseteq_P$ is more permissive than $\sqsubseteq$. Moreover, negating the statement of the proposition (if $L_1 \not\sqsubseteq_P L_2$ then $L_1 \not\sqsubseteq_{P'} L_2$) establishes that if exercising a privilege P does not allow for the flow of information from L_1 to L_2, then exercising a privilege delegated from P will also fail to allow the flow. This property is especially useful in guaranteeing soundness of privilege separation.

4 Model Extensions

The base DC label model, as described in Section 2, can be used to implement complex DIFC systems, despite its simplicity. Furthermore, the model can easily be further extended to support features of existing security (IFC and capability) systems, as we detail below.

4.1 Principal hierarchy

As previously mentioned, DLM [21] has a notion of a principal hierarchy defined by a reflexive and transitive relation, called *acts for*. Specifically, a principal p can act for another principal p', written $p \succeq p'$, if p is at least as powerful as p': p can read, write, declassify, and endorse all objects that p' can; the principal hierarchy tracks such relationships.

To incorporate this feature, we modify our model by encoding the principal hierarchy as a set of axioms Γ. Specifically, if $p \succeq p'$, then $(p \implies p') \in \Gamma$. Consequently, Γ is used as a hypothesis in every proposition. For example, without the principal hierarchy $\emptyset \vdash p_1 \implies [p_2 \vee p_3]$ does not hold, but if $p_1 \succeq p_2$ then $(p_1 \implies p_2), \Gamma \vdash p_1 \implies [p_2 \vee p_3]$ does hold. We, however, note that our notion of privileges and label component clauses (disjunction categories) can be used to capture such policies, that are expressible in DLM only through the use of the principal hierarchy. Compared to DLM, DC labels can be used to express very flexible policies (e.g., joint ownership) even when $\Gamma = \emptyset$.

4.2 Using DC labels in a distributed setting

For scalability, extending a system to a distributed setting is crucial. Addressing this issue, Zeldovich et al. [31], provide a distributed DIFC system, called DStar. DStar is a framework (and protocol) that extends OS-level DIFC systems to a distributed setting. Core to DStar is the notion of an *exporter* daemon, which, among other things, maps DStar network labels to OS local labels such as DC labels, and conversely. DC labels (and privileges) are a generalization of DStar labels (and privileges)—the core difference being the ability of DC labels to represent joint ownership of a category with disjunctions, a property expressible in DStar only with privileges. Hence, DC labels can directly be used when extending a system to a distributed setting. More interestingly, however, we can extend DStar, while remaining backwards compatible (since every DStar label can be expressed using a DC label), to use disjunction categories and thus, effectively, use DC labels as network labels—this extension is part of our future work.

4.3 Delegation and pseudo-principals

As detailed in Section 2.1, our decentralized privileges can be delegated and thus create a privilege hierarchy. Specifically, a process with a set of privileges may delegate a category it owns (in the form of a single-category privilege), which can then be further *granted* or delegated to another process.

In scenarios involving delegated privileges, we introduce the notion of a *pseudo-principal*. Pseudo-principals allows one to express providence on data, which is particularly useful in identifying the contributions of different computations to a task. A pseudo-principal is simply a principal (distinguished by the prefix #) that cannot be owned by any piece of code and can only be created when a privilege is delegated. Specifically, a process that owns principal p may delegate a single-category privilege $\{[p \vee \#c]\}$ to a piece of code c. The disjunction is used to indicate that the piece of code c is responsible for performing a task been delegated by the code owing p, which also does not trust c with the privilege p. Observe that the singleton $\{\#c\}$ cannot appear in any privilege, and as a result, if some data is given to p with the integrity restriction $[p \vee \#c]$, then the piece of code c must have been the originator. In a system with multiple components, using pseudo-principals, one can enforce a pipeline of operations, as shown by the implementation of a mail delivery agent in Section 6.

We note that pseudo-principals are treated as ordinary principals in label operations. Moreover, in our implementation, the distinction is minimal: principals are strings that cannot contain the character '`#`', while pseudo-principals are strings that always have the prefix '`#`'.

4.4 Privilege revocation

In dynamic systems, security policies change throughout the lifetime of the system. It is common for new users to be added and removed, as is for privileges to be granted and revoked [2]. Although our model can be extended to support revocation similar to that of public key infrastructures [11], we describe a selective revocation approach, common to capability-based systems [24].

To allow for the flexibility of selective revocation, it is necessary to keep track of a delegation chain with every category in a delegated privilege. For example, if processes A and C respectively delegate the single-principal privileges $\{a\}$ and $\{c\}$ to process B, B's privilege will be encoded as $\{(\{A \to B\}, a), (\{C \to B\}, c)\}$. Similarly, if B delegates $\{[a \vee c]\}$ to D, the latter's privilege set will be $\{(\{A \to B \to D, C \to B \to D\}, [a \vee c])\}$. Now, to selectively revoke a category, a process updates a system-wide revocation set Ψ with a pair consisting of the chain prefix and a privilege (it delegated) to be revoked. For example, A can revoke B's ownership of $\{a\}$ by adding $(\{A \to B\}, a)$ to Ψ. Consequently, when B or D perform a label comparison involving privileges, i.e., use $\sqsubseteq_P$, the revocation set Ψ is consulted: since $A \to B$ is a prefix in both cases, and $a \implies a$ and $a \implies [a \vee c]$, neither B nor D can exercise their delegated privileges. More generally, ownership of single-category privilege $\{c\}$ with chain x is revoked if there is a pair $(y, \psi) \in \Psi$ such that the chain y is a prefix of a chain in x and $\psi \implies c$. We finally note that, although this description of revocation relies on a centralized revocation set Ψ, selective revocation, in practice, can be implemented without a centralized set, using patterns such as Redell's "caretaker pattern" [24, 18] with wrapper, or membrane, objects transitively applying the revocation [19, 18].

5 Security Labeling Patterns

When building practical IFC systems, there are critical design decisions involving: (1) assigning labels to entities (data, channels, etc.), and (2) delegating privileges to executing code. In this section, we present *patterns* that can be used as a basis for these design decisions, illustrated using simplified examples of practical system applications.

5.1 Confinement and access control

A very common security policy is *confinement*: a program is allowed to compute on sensitive data but cannot export it [16, 26]. The tax-preparation example of Section 2 is a an examples of a system that enforces confinement.

In general, we may wish to confine a computation and guarantee that it does not release (by declassification) user A's sensitive data to the public network or any other channel. Using the network as an illustrative example, and assuming A's sensitive data is labeled L_A, confinement may be achieved by executing the computation with privileges P chosen such that $L_A \not\sqsubseteq_P L_{\text{pub}}$. A complication is that most existing IFC systems (though not all, see, e.g., [6, 14]) are susceptible to covert channel attacks that circumvent the restrictions based on labels and privileges. For example, a computation with no privileges might read sensitive data and leak information by, e.g., not terminating or affecting timing behavior. To address confinement in the presence of covert channels, we use the notion of *clearance* [5], previously introduced and formalized in [30, 30, 28] in the context of IFC.

Clearance imposes an upper bound on the sensitivity of the data that the computation can read. To prevent a computation from accessing (reading or writing) data labeled L_A, we set the computation's clearance to some L_C such that $L_A \not\sqsubseteq L_C$. With this restriction, the computation may read data labeled L_D only if $L_D \sqsubseteq L_C$. Observe that in a similar manner, clearance can be used to enforce other forms of discretionary access control.

5.2 Privilege separation

Using delegation, a computation may be compartmentalized into sub-computations, with the privileges of the computation subdivided so that each sub-computation runs with *least privilege*. Consider, for example, a privilege-separated mail delivery agent (MDA) that performs spam filtering.

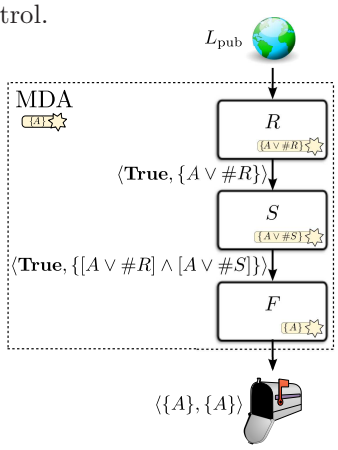

Fig. 2: Simple MDA.

As with many real systems, the example MDA of Fig. 2 is composed of different, and possibly untrustworthy, modules. In this example, the components are a network receiver, R, and a spam filter, S. Instead of combining the components into a monolithic MDA, the MDA author can segregate the untrustworthy components and execute then with the principle

of least privilege. This avoids information leaks and corruption due to negligence or malfeasance on the component authors' part. Specifically, the receiver R is executed with the delegated privilege $\{[A \vee \#R]\}$, and the spam filter S is executed with the privilege $\{[A \vee \#S]\}$. As a consequence, R and S cannot read A's sensitive information and leak it to the network, corrupt A's mailbox, nor forge data on A's behalf.

Additionally, the MDA can enforce the policy that a mail message always passes through both receiver R and spam filter S. To this end, the MDA includes a small, trusted forwarder F, running with the privilege $\{A\}$, which endorses messages on behalf of A and writes them to the mailbox only after checking that they have been endorsed by both R and S. In a similar manner, this example can be further extended to verify that the provenance of a message is the network interface, or that the message took a specific path (e.g., R then S, but not S then R), among other.

5.3 User authentication

Another common requirement of security systems is user authentication. We consider password-based login as an example, where a successful authentication corresponds to granting the authenticated user the set of privileges associated with their credentials. Furthermore, we consider authentication in the context of (typed) language-level DIFC systems; an influential OS-level approach has been considered in [30]. Shown in Fig. 3 is an example system which consists of a login client L, and an authentication service A_U.

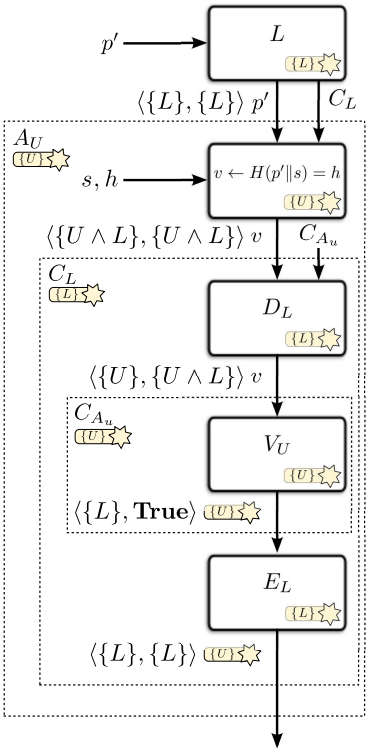

Fig. 3: User authentication.

To authenticate user U, the login client *invokes* the user authentication service A_U, which runs with the $\{U\}$ privilege. Conceptually, when invoked with U's correct credentials, A_U grants (by delegating) the caller the $\{U\}$ privilege. However, in actuality, the login client and authentication service are in mutual distrust: L does not trust A_U with U's password, for A_U might be malicious and simply wish to learn the password, while A_U does not trust L to grant it the $\{U\}$ privilege without first verifying credentials. Consequently, the authentication requires several steps.

We note that due to the mutual distrust, the user's stored salt s and password hash $h = H(p\|s)$ is labeled with both, the user and login client's, principals, i.e., h and s have label $\langle\{U \wedge L\}, \{U \wedge L\}\rangle$. Solely, labeling them $\langle\{U\}, \{U\}\rangle$ would allow A_U to carry out an off-line attack to recover p. The authentication procedure is as follows.

1. The user's input password p' to the login client is labeled $\langle\{L\}, \{L\}\rangle$, and along with a closure C_L is passed to the authentication service A_U. As further detailed below, closures are used in this example as a manner to exercise privileges under particular conditions and operations.
2. A_U reads U's stored salt s and password hash h. It then computes the hash $h' = H(p'\|s)$ and compares h' with h. The label of this result is simply the join of h and h': $\langle\{U \wedge L\}, \{L\}\rangle$. Since A_U performed the computation, it endorses the result by adding U to its integrity component; for clarity, we name this result v, as show in Fig 3.

- *Remark:* At this point, neither L nor A_U are able to read and fully declassify the secret password-check result v. Moreover, without eliminating the mutual distrust, neither L nor S can declassify v directly. Consider, for example, if A_U is malicious and had, instead, performed a comparison of $H(p'\|s)$ and $H(p''\|s)$, for some guessed password p''. If L were to declassify the result, A_U would learn that $p = p''$, assuming the user typed in the correct password, i.e., $p = p'$. Hence, we rely on purely functional (and statically-typed) closures to carry out the declassification indirectly.

3. When invoking A_U, L passed a declassification closure C_L, which has the $\{L\}$ privilege locally bound. Now, A_U invokes C_L with v and its own declassification closure C_{A_U}.
4. C_L declassifies v (D_L in Fig. 3) to $\langle\{U\}, \{U \wedge L\}\rangle$, and then invokes C_{A_U} with the new, partially-declassified result.
5. The C_{A_U} closure has the $\{U\}$ privilege bound and upon being invoked, simply verifies the result and its integrity (V_U in Fig. 3). If the password is correct v is true and then C_{A_U} returns the privilege $\{U\}$ labeled with $\langle\{L\}, \mathbf{True}\rangle$; otherwise it returns the empty privilege set. It is important that the integrity of v be verified, for a malicious L could provide a closure that forges password-check results, an attempt to wrongfully gain privileges.
6. The privilege returned from invoking C_{A_U} is endorsed by C_L (E_L in Fig. 3), only if its secrecy component is L. This asserts that upon returning the privilege from C_L, A_U cannot check if the privilege is empty or not, and thus infer the comparison result.
7. It only remains for A_U to forward the labeled privilege back to L.

We finally note that the authentication service is expected to keep state that tracks the number of attempts made by a login client, as each result leaks a bit of information; to limit the number of unsuccessful attempts requires the use of a (minimal) code that is trusted by both L and A_U, as shown in [30].

6 Implementing DC labels

We present two Haskell implementations of DC labels[3]. The first, `dclabel`, is a library that provides a simple embedded domain specific language for constructing and working with dynamic labels and privileges. Principals in the `dclabel`

[3] Available at `http://www.scs.stanford.edu/~deian/dclabels`

library are represented by strings, while label components are lists of clauses (categories), which, in turn, are lists of principals. We use lists as sets for simplicity and because Haskell supports list comprehension; this allowed for a very simple translation from the formal definitions of this paper to (under 180 lines of) Haskell code. We additionally implemented the instances necessary to use DC labels with the label-polymorphic dynamic DIFC library, LIO [28]. Given the simplicity of the implementation, we believe that porting it to other libraries, such as [17, 13], can be accomplished with minimal effort. Finally, we note that our implementation was thoroughly tested using the QuickCheck[4] library, however formal verification of the implementation using Zeno [27], a Haskell automated proof system, was unsuccessful. This is primarily due to Zeno's infancy and lack of support for analyzing Haskell list comprehension. A future direction includes implementing DC labels in Isabelle or Coq from which a provably-correct Haskell implementation can be extracted.

Although we have primarily focused on dynamic IFC, in cases where covert channels, runtime overhead, or failures are not tolerable, DC labels can also be used to enforce IFC statically. To this end, we implement `dclabel-static`, a prototype IFC system that demonstrates the feasibility of statically enforcing DIFC using secrecy-only DC labels, without modifying the Haskell language or the GHC compiler. Since DC labels are expressed using propositional logic, a programming language that has support for sum, product, and function types can be used, *without modification*, to enforce information flow control according to the Curry-Howard correspondence [12, 9]. According to the correspondence, disjunction, conjunction and implication respectively correspond to sum, product, and function types. Hence, for a secrecy-only DC label, to prove $L_1 \sqsubseteq L_2$, i.e., $L_2 \implies L_1$, we need only construct a function that has type $L_2 \to L_1$: successfully type-checking a program directly corresponds to verifying that the code does not violate IFC.

The library exports various type classes and combinators that facilitates the enforcement of static IFC. For example, we provide type constructors to create labels from principals—a principal in this system is a type for which an instance of the `Principal` type class is defined. To label values, we associate labels with types. Specifically, a labeled type is a wrapper for a product type, whose first component is a label, and whose second component, the value, cannot be projected without declassification. The library further provides a function, `relabel`, which, given a labeled value (e.g., $(L_1, 3)$), a new label L_2, and a proof of $L_1 \sqsubseteq L_2$ (a lambda term of type $L_2 \to L_1$), returns the relabeled value (e.g., $(L_2, 3)$). Since providing such proofs is often tedious, we supply a tool called `dcAutoProve`, that automatically inserts proofs of can-flow-to relations for expressions named `auto`, with an explicit type signature. Our automated theorem prover is based a variant of Gentzen's LJ sequent calculus [7].

[4] `http://hackage.haskell.org/package/QuickCheck`

7 Related work

DC labels closely resemble DLM labels [21] and their use in Jif [22]. Like DC labels, DLM labels express both secrecy and integrity policies. Core to a DLM label are components that specify an owner (who can declassify the data) and a set of readers (or writers). Compared to our disjunction categories, DLM does not allow for joint ownership of a component—they rely on a centralized principal hierarchy to express partial ownership. However, policies (natural to DLM) which allow for multiple readers, but a single owner, in our model, require a labeling pattern that relies on the notion of clearance, as discussed in Section 5 and used in existing DIFC systems [30, 31, 28]. Additionally, unlike to DLM labels as formalized in [20], DC labels form a bounded lattice with a join and meet that respectively correspond to the least upper bound and greatest lower bound; the meet for DLM labels is not always the greatest lower bound.

The language Paralocks [3] uses Horn clauses to encode fine-grained IFC policies following the notion of locks: certain flows are allowed when corresponding locks are open. Constraining our model to the case where a privilege set is solely a conjunction of principals, Paralocks be easily used to encode our model. However, it remain an open problem to determine if disjunctive privileges can be expressed in their notion of *state*.

The Asbestos [8] and HiStar [30] operating systems enforce DIFC using Asbestos labels. Asbestos labels use the notion of categories to specify information flow restrictions in a similar manner to our clauses/categories. Unlike DC labels, however, Asbestos labels do not rely on the notion of principals. We can map a subset of DC labels to Asbestos labels by mapping secrecy and integrity categories to Asbestos levels **3** and **0**, respectively. Similarly ownership of a category maps to level $\star$. This mapping is limited to categories with no disjunction, which are equivalent to DStar labels [31], as discussed in Section 4. Mapping disjunction categories can be accomplished by using the system's notion of privileges. Conversely, both Asbestos and DStar labels are subsumed by our model. Moreover, compared to these systems we give precise semantics, prove soundness of the label format, and show its use in enforcing DIFC statically.

Capability-based systems such as KeyKOS [1], and E [19] are often used to restrict access to data. Among other purposes, capabilities can be used to enforce discretionary access control (DAC), and though they can enforce MAC using patterns such as membranes, the capability model is complimentary. For instance, our notion of privilege is a capability, while a delegated privilege loosely corresponds to an attenuated capability. This notion of privileges as capabilities is like that of Flume [15]. However, whereas they consider two types of privilege (essentially one for secrecy and another for integrity), our notion of privilege directly corresponds to ownership and conferring the right to exercise it in any way. Moreover, delegated privileges and the notion of disjunction provides an equal abstraction.

8 Conclusion

Decentralized information flow control can be used to build applications that enforce end-to-end security policies using untrusted code. DIFC systems rely on labels to track and enforce information flow. We present disjunction category labels, a new label format useful in enforcing information flow control in systems with mutually distrusting parties. In this paper, we give precise semantics for DC labels and prove various security properties they satisfy. Furthermore, we introduce and prove soundness of decentralized privileges that are used in declassifying and endorsing data. Compared to Myers and Liskov's DLM, our model is simpler and does not rely on a centralized principal hierarchy, our privilege hierarchy is distributed. We highlight the expressiveness of DC labels by providing several common design and labeling patterns. Specifically, we show how to employ DC labels to express confinement, access control, privilege separation, and authentication. Finally, further illustrating flexibility of the model, we describe two Haskell implementations: a library used to perform dynamic label checks, compatible with existing DIFC systems, and a prototype library that enforces information flow statically by leveraging Haskell's module and type system.

Acknowledgments This work was supported by DARPA CRASH and PROCEED, Google, the Swedish research agency VR, the NSF, and the AFOSR. D. Stefan is supported by the DoD through the NDSEG Fellowship Program.

References

1. A. C. Bomberger, A. P. Frantz, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The KeyKOS nanokernel architecture. In *Proc. of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, April 1992.
2. D. Boneh, X. Ding, G. Tsudik, and C. Wong. A method for fast revocation of public key certificates and security capabilities. In *Proceedings of the 10th conference on USENIX Security Symposium-Volume 10*, pages 22–22. USENIX Association, 2001.
3. N. Broberg and D. Sands. Paralocks: role-based information flow control and beyond. In *SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 431–444, 2010.
4. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.
5. Department of Defense. *Trusted Computer System Evaluation Criteria (Orange Book)*, DoD 5200.28-STD edition, December 1985.
6. D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *2010 IEEE Symposium on Security and Privacy*, pages 109–124. IEEE, 2010.
7. R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, pages 795–807, 1992.
8. P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *Proc. of the 20th ACM Symposium on Operating Systems Principles*, pages 17–30, Brighton, UK, October 2005. ACM.
9. J. Gallier. Constructive logics part i: A tutorial on proof systems and typed λ-calculi. *Theoretical computer science*, 110(2):249–339, 1993.
10. J. Goguen and J. Meseguer. Security policies and security models. In I. C. S. Press, editor, *Proc of IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.

11. C. Gunter and T. Jim. Generalized certificate revocation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 316–329. ACM, 2000.
12. W. Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, pages 479–490, 1980.
13. M. Jaskelioff and A. Russo. Secure multi-execution in Haskell. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer Verlag, June 2011.
14. V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing-and termination-sensitive secure information flow: Exploring a new approach. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 413–428. IEEE, 2011.
15. M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proc. of the 21st Symp. on Operating Systems Principles*, October 2007.
16. B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.
17. P. Li and S. Zdancewic. Arrows for secure information flow. *Theoretical Computer Science*, 411(19):1974–1994, 2010.
18. M. Miller and J. Shapiro. Paradigm regained: Abstraction mechanisms for access control. *Advances in Computing Science–ASIAN 2003*, pages 224–242, 2003.
19. M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.
20. A. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *IEEE Security and Privacy, 1998.*, pages 186–197. IEEE, 1998.
21. A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, 1997.
22. A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. on Computer Systems*, 9(4):410–442, October 2000.
23. C. Papadimitriou. *Complexity Theory*. Addison Wesley, 1993.
24. D. Redell and R. Fabry. Selective revocation of capabilities. In *Proceedings of the International Workshop on Protection in Operating Systems*, pages 192–209, 1974.
25. A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 255–269, June 2005.
26. J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. of the IEEE*, 63(9):1278–1308, September 1975.
27. W. Sonnex, S. Drossopoulou, and S. Eisenbach. Zeno: A tool for the automatic verification of algebraic properties of functional programs. Technical report, Imperial College London, Feb. 2011.
28. D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*, pages 95–106. ACM SIGPLAN, September 2011.
29. S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 15–23, June 2001.
30. N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation*, pages 263–278, Seattle, WA, November 2006.
31. N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *Proc. of the 6th Symp. on Networked Systems Design and Implementation*, pages 293–308, San Francisco, CA, April 2008.