# IFDB: Decentralized Information Flow Control for Databases

David Schultz[†] Barbara Liskov

MIT CSAIL

{das,liskov}@csail.mit.edu

## Abstract

Numerous sensitive databases are breached every year due to bugs in applications. These applications typically handle data for many users, and consequently, they have access to large amounts of confidential information.

This paper describes IFDB, a DBMS that secures databases by using decentralized information flow control (DIFC). We present the Query by Label model, which introduces new abstractions for managing information flows in a relational database. IFDB also addresses several challenges inherent in bringing DIFC to databases, including how to handle transactions and integrity constraints without introducing covert channels.

We implemented IFDB by modifying PostgreSQL, and extended two application environments, PHP and Python, to provide a DIFC platform. IFDB caught several security bugs and prevented information leaks in two web applications we ported to the platform. Our evaluation shows that IFDB's throughput is as good as PostgreSQL for a real web application, and about 1% lower for a database benchmark based on TPC-C.

***Categories and Subject Descriptors*** H.2.4 [*Database Management*]: Systems

***General Terms*** Design, Security

***Keywords*** information flow control, DIFC

## 1. Introduction

Breaches of online systems resulting in the exposure of sensitive databases continue to occur, despite advances in application security. Large-scale breaches have been enabled by SQL injection attacks, omitted authentication checks, and scripts that inadvertently reveal more information than they should.

There has been much recent interest in using decentralized information flow control (DIFC) [25] to improve application security [6, 11, 18, 36]. Unlike access control, where checks for authorization are made when data are read or written, information flow control systems track data as they flow through the system and restrict what can be released. DIFC extends earlier work on information flow control [2, 9] to protect data for many users, each with a distinct policy. There is a gap in prior research, however: despite the fact that many applications store sensitive data in relational databases, none of the prior work has developed a comprehensive model for DIFC in databases.

This paper introduces IFDB, a new approach to securing sensitive databases based on DIFC. IFDB is intended to work alongside DIFC programming languages and operating systems: it supports a comprehensive approach to information flow control that tracks flows and enforces a security policy both in the DBMS and the application platform.

CarTel [14], one of the applications we studied, illustrates the power of IFDB. CarTel collects information from GPS-equipped cars and provides users with maps and statistics about their past drives and the drives of their friends. By converting CarTel to use IFDB, we substantially reduced the amount of application code that had to be trusted for confidentiality, and also fixed several bugs.

Our version of CarTel annotates data as they arrive, e.g., as being location data for Alice. These annotations, together with the tracking provided by information flow control, prevent unintended disclosure. For example, IFDB can enforce Alice's policy that only she can see her current location, and only she and her friends can see her past drives.

CarTel produces drive data from location measurements using a combination of complex stored procedures and application procedures. The procedures require access to the location measurements, so an access control policy could not prevent bugs in the code from compromising users' privacy. Information flow control prevents the code from releasing location measurements inappropriately, and it does so for both the stored procedures and application procedures.

Integrating DIFC into a relational database presents several new challenges. First, the standard relational query model

Copyright © ACM, 2013. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in EuroSys'13, April 15–17, 2013, Prague, Czech Republic, http://doi.acm.org/10.1145/2465351.2465357.

*Eurosys'13* April 15–17, 2013, Prague, Czech Republic

Copyright © 2013 ACM 978-1-4503-1994-2/13/04. . . $15.00

[†] Current affiliation: Google, Inc.

doesn't provide a good basis for reasoning about flows of sensitive information; for example, a query for records about hospital patients who don't have cancer can reveal indirectly which patients do have cancer. Second, the DBMS must provide ways to manage the information flows that arise through mechanisms such as complex queries, stored procedures, and views. Third, important database features such as transactions and constraints can lead to information leaks without appropriate precautions.

IFDB addresses these challenges as follows.

— IFDB uses the *Query by Label* model, which provides a practical way to do relational queries while respecting information flow rules. Query by Label extends earlier work on multi-level-secure databases, such as SeaView [21], to support DIFC.

— To manage information flows in the database, IFDB provides *declassifying views* and *stored authority closures*.

— To make transactions and constraints safe, IFDB introduces several new concepts such as *transaction commit labels* and *DECLASSIFYING clauses*.

Additionally, IFDB is easy to use. It works with existing languages such as SQL, PHP, and Python, with straightforward extensions to support DIFC. Confidentiality policies are specified in terms of delegation and exercise of authority—a model programmers are familiar with. Furthermore, IFDB is the first system to integrate a uniform set of abstractions for managing information flows into both the programming language and the DBMS.

To show that IFDB is practical, we converted two existing web applications to use it: the CarTel system introduced earlier, and HotCRP [17], the conference management system used for this conference. IFDB prevented several information leaks in both applications, and we found that it was much easier to verify security properties in the IFDB-enabled applications than in the original programs. Furthermore, using DIFC did not noticeably affect the scalability of the DBMS running CarTel, while a TPC-C-like benchmark of the database alone showed a 1% reduction in throughput. In addition, our experience with IFDB provided insight into how to use IFDB in other applications; we describe a methodology for application development in Section 6.4.

## 2. Architecture and Trusted Base

IFDB tracks information as it flows through the database and through the applications that interact with the database. On the database side, IFDB only accepts connections from applications running within a trusted runtime environment that tracks and enforces information flow control. We have created two such environments by implementing modest extensions to PHP and Python.

Figure 1 shows a deployment of IFDB and PHP-IF running the CarTel application. Since IFDB and the application platform work together to enforce the information flow policy,

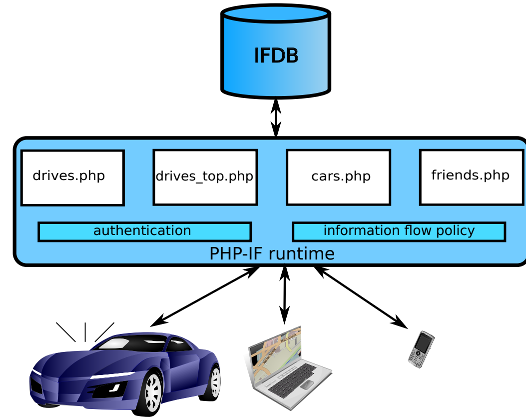


**Figure 1.** An IFDB deployment with the CarTel application. The blue shaded regions are trusted components. CarTel receives data from GPS transponders in vehicles, and vehicle owners interact with CarTel via a web interface.

they are both part of the trusted computing base for the system. As illustrated in the figure, the components of the application that authenticate external users and define the information flow policy itself are also essential for security. However, the remainder of the application code, including SQL stored procedures, is *not* trusted.

IFDB tracks information flows on a per-process granularity within the application platform (where fine-grained tracking would be expensive) and fine-grained, per-tuple tracking only within the database. The database is the primary shared medium, through which leaks could occur, so fine-grained tracking within the database is essential.

## 3. Information Flow Model

This section describes the information flow model used in IFDB; Sections 4 and 5 explain how we extend the database to support this model. We have chosen to base our work on the model provided by the Aeolus DIFC platform [6]; we discuss alternatives in Section 9.

### 3.1 Tags and Labels

IFDB uses *tags*, which are identifiers attached to data to denote their sensitivity. For example, the alice-location tag might represent the secrecy concerns of Alice's GPS coordinates, while the bob-contact tag is for Bob's contact information.

*Labels* are sets of tags. Each data object has a label that summarizes the sensitivity of all the data it contains. Labels of data objects are immutable; they are specified when the object is created and cannot be changed later. Each process also has a label, which expands over time to reflect the sensitivity of all the data that has affected the process. Conceptually, a process becomes "contaminated" by the labels of all the data it reads.

Some computations, such as one that computes statistics about all users' driving histories, run over data with many dif-

ferent tags. IFDB provides convenience and representational efficiency for such computations through *compound tags*, which can be used to group tags so that they can be used as a unit. For example, the alice-location tag is a member of the all-locations compound tag. A tag is declared as a member of one or more compounds when it is created. At present, IFDB does not allow the links between a tag and its compounds to change, since changing them would effectively relabel all data protected by that tag.

In addition to the *secrecy* labels described above, IFDB supports *integrity* labels, which make it possible to track whether data came from trusted sources. We do not discuss integrity labels in this paper; see [29] for the details.

### 3.2 Controlling Information Flow

IFDB ensures that the label of each object reflects the tags of all the data that produced it, and the label of each process reflects the tags of all the data the process read. It does this by enforcing the following standard rule [2]:

**Information Flow Rule.** Information is permitted to flow from a source with label $L_S$ to a destination with label $L_D$ provided that $L_S \subseteq L_D$.

The rule also controls what information can be released. The outside world is treated as having an empty label. This implies that in order to send anything to the outside world, e.g., to a web client, a process must have an empty label.

Since contaminated processes can't release information, these rules by themselves would make it impossible to get any sensitive data out of the system. Hence, it's necessary to have a way to remove tags from labels. Removing a tag is called *declassification*, and it has two main uses. First, a process might declassify after transforming or summarizing data to remove confidential information, e.g., computing the average speed of all CarTel users on a road. Second, a process might declassify to send sensitive information to an authorized user, e.g., to allow Alice to view her own CarTel driving history.

Declassification isn't safe in general because it removes constraints on information flow, so it requires authority. Specifically, the declassify($T$) operation, which removes tag $T$ from the invoking process' label, requires that the process have authority for $T$. Information flow policy in IFDB is specified by controlling the circumstances under which particular tags can be declassified.

Authority in IFDB is bound to *principals*, which are entities in the system with security interests, such as users and roles. Each process runs with the authority of a particular principal. For example, a web servlet might run with the authority of the user making the request.

Each tag in IFDB has an owning principal that has complete authority over that tag. Any principal can create a tag, and in doing so becomes the owner of that tag. Owners define the information flow policy for data covered by their tags through delegation and exercise of this authority. Authority can be delegated and given to users, application procedures (Section 3.3), and stored procedures and views (Section 4.3). For example, in a medical system, Bob can delegate authority for the tag on his medical record to his doctor. This allows the application, when acting on behalf of the doctor, to declassify Bob's medical record and send it to the doctor's web browser. Additionally, principals can revoke authority that they granted previously.

The authority state is itself an object with an empty label, so a process must have an empty label to make delegations or revocations. This restriction ensures that modifications to the authority state cannot be used as a covert channel.

### 3.3 The Principle of Least Privilege

Delegation of authority makes it possible to define *who* can declassify, but it doesn't constrain *how* that authority may be used. To that end, IFDB supports *reduced authority calls* and *authority closures*. Reduced authority calls allow the caller to run code with less authority. An authority closure is a procedure that is bound to a principal; it receives its authority when it is created, and the code that creates it must have the authority being granted. When the closure is called, it runs with this authority. For example, a procedure that computes traffic conditions can be bound to a principal that is authoritative for the location tags of every CarTel driver.

Both mechanisms support the Principle of Least Privilege [27] by making it easy to limit where authority can be used, which in turn makes it easier to reason about security, since only code with that authority must be considered.

Database systems traditionally have an administrator role who is responsible for setting up and managing the database. IFDB also limits the privilege of the administrator. An administrator must still define the database schema, but lacks the authority to declassify any tags.

## 4. Query By Label

This section introduces Query by Label, which allows us to extend the model introduced in Section 3 to a database.

### 4.1 Labels in the Database

IFDB uses labels to track information flow within the database. There are three plausible granularities at which data could be labeled: tables, tuples, and fields. IFDB attaches labels to tuples. Labeling tuples captures a common case—a table that contains information about many different users, each with separate privacy concerns. For example, the Drives table in CarTel contains information about all users' drives, but each tuple has a label specific to the respective driver.

Labeling fields allows different parts of a tuple to have different confidentiality, whereas labels in IFDB must reflect the contamination of all fields. However, in Section 4.4 we introduce *declassifying views*, which can be used to achieve the power of field-level labels. Tuple labels avoid the extra overhead of per-field labels provided in systems such as SeaView [21], and the consequent semantic problems [31].

## 4.2 Queries

In the standard relational model, a query over a table conceptually reads every tuple in the table. For example, consider the following query in a database with medical records:

```
SELECT * FROM PatientRecords
  WHERE condition <> 'cancer'
```

The query returns a list of patients in a clinic who *do not* have cancer; however, it implicitly reveals that all the patients who are not listed *do* have cancer. Therefore, the contamination associated with the query would necessarily include the labels of every tuple in the table.

Processes need a way to limit their contamination, however, because they will be unable to communicate if they are too contaminated. We provide the needed semantics using *Query by Label*. In our system each query has an associated label, which is the label of the process issuing the query. This label is used as a filter:

**Label Confinement Rule.** A query submitted by a process with label $L_P$ is performed on the subset of the database consisting of all tuples $T_i$ with labels $L_{T_i}$ such that $L_{T_i} \subseteq L_P$.

For reads, this constraint is simply an instantiation of the standard information flow rule (Section 3.1): A process should not see tuples whose contamination isn't covered by its own label. For example, consider the table illustrated in Figure 2 that stores medical records for patients with HIV. If a process with label {bob_medical} executes the query

```
SELECT * FROM HIVPatients
  WHERE patient_name = 'Bob'
   AND patient_dob = '6/26/78'
```

it will receive the tuple for Bob. However, if the process had an empty label, or a label {john_medical}, it would receive no tuples. Therefore, a process can read Bob's record only if its label is contaminated for Bob.

For writes, however, we need more: the confinement rule covers the information observed by the query, but in addition we need to define what tuples can be *modified* by the query. Our rule is:

**Write Rule.** A process with label $L_P$ can write a tuple with label $L_T$ only if $L_T \supseteq L_P$.

This rule is needed because otherwise a process contaminated with some secret would be able to write tuples whose labels didn't reflect that secret. Taken together, the two rules imply that all tuples written by the query have *exactly* label $L_P$. Of course, processes can change their labels over time, so in later queries, different tuples could be modified.

Thus, IFDB enforces the following restrictions for basic queries executed by a process with label $L_P$:

— SELECTs return only tuples $T_i$ such that $L_{T_i} \subseteq L_P$.

— INSERTs add tuples with exactly label $L_P$.

**Table HIVPatients**

| _label | patient_name | patient_dob | ... |
|---|---|---|---|
| {alice_medical} | Alice | 2/1/60 | ... |
| {bob_medical} | Bob | 6/26/78 | ... |
| {cathy_medical} | Cathy | 4/22/71 | ... |

Primary key (patient_name, patient_dob)

**Figure 2.** An example from a medical records system with a table containing specialized records for patients with HIV

— UPDATEs and DELETEs affect only tuples with label $L_P$. An attempt to update or delete lower-labeled tuples will fail, while tuples with other labels are invisible to the update and are unaffected by it.

Since inserts are confined to a subset of the database that contains only the tuples whose labels are contained in the label of the process, it is possible that an insert might add a tuple with the same primary key as one already in the table, but with a different label. We discuss our solution to this problem in Section 5.2.1.

Query by Label limits queries to tuples whose labels are subsets of the process label. However, processes can explicitly specify additional conditions on the label by referring to an immutable system column of type INT[] called _label, which exists in every table. Section 5.2.4 provides an example of how this can be useful.

Query by Label requires that all label changes are explicit: a process must set its label to control what tuples its reads, and the label of any tuple it writes. Explicit label changes prevent certain covert channels [36], and they ensure that code running with authority does not use that authority to release information accidentally.

## 4.3 Declassifying Views and Stored Procedures

IFDB introduces *declassifying views*, which are an adaptation of authority closures to the relational model; instead of binding authority to code, authority is bound to the definition of a view. This powerful idea permits a restricted view of a data set to have a different label than the original source data. Any user can create a declassifying view; however, the user must have whatever authority is being given to the view.

For example, tuples in the ContactInfo table in HotCRP are sensitive, but the list of program committee members, which is derived from a projection and selection on ContactInfo, ought to be public. Using declassifying views, we can define a PCMembers view that contains this information:

```
CREATE VIEW PCMembers AS
  SELECT firstName, lastName
    FROM ContactInfo
    WHERE IsPCMember(contactId)
  WITH DECLASSIFYING(all_contacts)
```

The view is defined to have authority for the all_contacts compound tag, and it uses its authority to declassify the tags in the base relation.

In [29], we describe extensions to support more sophisticated declassifying views: for instance, a view that extracts billing data from medical records must replace the *p*_medical tag with *p*_billing for each patient *p*. Additionally, we explain how to make declassifying views updatable using rewrite rules, a mechanism IFDB inherits from PostgreSQL.

IFDB also supports binding authority to code. Applications can use SQL stored procedures to run complex computations within the database system, and stored procedures in IFDB normally run with the authority of the caller. *Stored authority closures*, however, have authority bound in; again the creator of the closure must have the authority being bound to the closure. When an authority closure runs, it runs with the authority bound to it. Stored authority closures extend the authority closures described in Section 3.3 to the DBMS.

### 4.4 Data Independence

Within the IFDB database, data are labeled at the granularity of tuples. However, we would like to avoid constraining programmers to normalize their schemas according to the confidentiality concerns of the data. For example, suppose that users' payment information and contact information have separate tags and security policies. The programmer should be able to refer to the payment and contact information as separate relations or as a single relation.

IFDB achieves data independence through a combination of standard views and declassifying views (see Section 4.3). Standard views can be used to join two relations and treat them as one. For example, if a PaymentContact view is constructed using the standard outer join operator on the Payment and Contact tables, a process whose label contains only payment tags will see NULLs in place of the contact-related fields it isn't allowed to see. Such views can be used to simulate field-level labels, with semantics similar to the SeaView model [21].

Declassifying views are needed when we wish to store data with different labels together in a single tuple, for instance, in a physical PaymentContact table. To ensure that the data are not leaked, PaymentContact tuples must be tagged with both payment and contact tags. Logically, however, the payment tags cover some columns, whereas the contact tags cover other columns. Thus, we construct a declassifying view called Payment that projects the payment-related columns and declassifies the contact tags, and a declassifying view called Contact that projects the contact-related columns and declassifies the payment tags.

## 5. Transactions and Constraints

This section explains how IFDB addresses the challenges that transactions and integrity constraints pose for a DIFC system.

### 5.1 Transactions

Processes need to change their labels in mid-transaction so they can write tuples with different labels as part of a single transaction. For example, a user's contact information may have a different label than the user's password, but both should be added to the database in the same transaction when a new user is added to the system.

However, such a label change could allow a low-authority process to leak confidential information, as illustrated in the following example.

```
BEGIN
  INSERT INTO Foo VALUES ('Alice has HIV');
  PERFORM addsecrecy(alice_medical);
  SELECT * FROM HIVPatients WHERE pname='Alice';
  IF NOT FOUND THEN ABORT; END IF;
COMMIT;
```

The transaction first writes the string "Alice has HIV" with an empty label. Then it raises its label, checks whether Alice has HIV, commits if Alice has HIV and aborts otherwise. The result is that the string "Alice has HIV" is written with an empty (public) label only if Alice actually has HIV, which leaks Alice's medical information.

To prevent the problem, IFDB introduces *transaction commit labels*. A process is allowed to commit a transaction only if its label at the commit point is no more contaminated than any tuple in its write set. In the preceding example, the rule ensures that the transaction cannot commit and write Foo with an empty label unless the process were to exercise proper authority to declassify alice_medical. The underlying intuition is that all writes happen at the commit point, so that is where the information flow rules should apply.

Additionally, concurrency conflicts between transactions can lead to covert channels, so another rule is needed for serializable transactions. The *transaction clearance* rule says that a process running a transaction is allowed to add a tag to its label only if it is authoritative for that tag. (Our prototype is based on a DBMS that uses snapshot isolation, which is slightly weaker than serializability and doesn't require this restriction. Details can be found in [29].)

### 5.2 Integrity Constraints

Integrity constraints are desirable since they can suppress many application errors. However, problems arise when constraints involve tuples with different labels. The next two subsections explain how IFDB handles two important types of constraints in databases: uniqueness constraints and referential constraints. Section 5.2.3 discusses more general constraints, and Section 5.2.4 introduces label constraints.

#### 5.2.1 Uniqueness Constraints

Uniqueness constraints require that all tuples in a table have unique values for a particular column or combination of columns. The *primary key* for a table is always subject to a uniqueness constraint.

It is easy to check that a write obeys a constraint when all tuples needed to verify the constraint are visible to the process performing the write. However, a problem arises when the

question of whether the data conforms with the constraint depends on tuples the process should not be allowed to see.

We use the HIVPatients table in Figure 2 to construct an example of the problem. Consider the following inserts into that table:

1. Insert (Dan, 8/12/69) into HIVPatients with any label
2. Insert (Alice, 2/1/60) into HIVPatients with label {alice_medical}
3. Insert (Alice, 2/1/60) into HIVPatients with label { }

The first insert doesn't violate the constraint because there is no entry for Dan in the table; hence, it should succeed regardless of the label used. It is clear that the second insert violates the constraint because Alice already has an entry in the table, but enforcing the constraint and causing the second insert to fail reveals nothing, because the conflicting tuple is already visible to the process performing the insert. The third insert is the problematic one. Like the second insert, it violates the constraint; however, the process has an empty label, so it isn't supposed to see the conflicting tuple with a higher label.

Disallowing the insert when the constraint is violated leaks information about the existence of the tuple that can't be viewed, e.g., the process could learn that Alice is a patient in the clinic. Instead, IFDB uses *polyinstantiation* [21], which permits inserts of tuples that conflict with higher-labeled tuples. Clients running with lower labels, unaware of the higher-labeled tuples, see a consistent view of the database and are unaffected by the higher-labeled data. Clients running with higher labels, however, will see both tuples, distinguished only by their labels—a violation of the uniqueness constraint.

Polyinstantiation can lead to confusion: What does it mean to have two patients with identical primary keys but different labels? Much prior work is concerned with the use of polyinstantiation as an important feature in its own right, and proposes *cover stories* and different subjective opinions of the truth as possible answers to this question [28]. We advocate a simpler interpretation: polyinstantiated tuples are seen as mistakes. Since it would leak information to expose the mistakes to clients with lower labels (i.e., by notifying them of the conflict), IFDB instead exposes the mistakes to the clients with higher labels.

Applications that are not prepared to cope with multiple records when only one is expected can request an exact label (see Section 4.2) to hide erroneous tuples (e.g., tuples for Bob that don't have bob-tag in their label). This is effective because polyinstantiated tuples must have different labels. IFDB also supports *label constraints* (Section 5.2.4), which provide a way to prevent polyinstantiation.

### 5.2.2 Foreign Key Constraints

Foreign key constraints enforce a many-to-one mapping, or referential integrity, between a *referencing table* and a *referenced table*. For example, in CarTel, the carid field of the Drives table should refer to a valid car in the Cars table. Similarly, user ids in the Friends table should refer to valid users in the Users table.

Inserts in the referencing table can leak information because they allow the inserter to learn about the presence of the related tuple in the referenced table. Also deletes of tuples in the referenced table can allow the deleter to learn about the presence of tuples in the referencing table. The information exposed by these inserts and deletes isn't necessarily a problem; for example this is the case in the CarTel examples given above. However, consider the following:

— **Inserts.** Suppose every tuple in the HIVRecords table is constrained to refer to a patient listed in the HIVPatients table in Figure 2. Then a process running with an empty label could learn whether a particular patient is in HIVPatients by inserting a tuple in HIVRecords, since this cannot succeed for non-HIV patients.

— **Deletes.** The HIVPatients table itself might refer to another table, PatientContact. If the DBMS disallowed deletion of a patient's contact information only if he has a referring tuple in HIVPatients, that provides an effective (albeit destructive) means to determine which patients have HIV.

IFDB solves both problems by constraining inserts, using the following rule:

**Foreign Key Rule.** To insert a tuple $A$ that refers to a tuple $B$ under a foreign key constraint, the requesting process must have authority to declassify for each tag in the symmetric difference of the two tuples' labels, $L_A \ominus L_B$, and must specify these tags explicitly. (The symmetric difference is all the tags that appear in one label but not the other.)

This rule prevents both problems described above. In the case of inserts, it acknowledges that the inserter is in fact reading from the referenced table and thus becomes contaminated by doing the insert. Therefore, it allows the read only if the inserter has authority over the additional tags in this contamination. Allowing the insert under these conditions is safe, because the inserter could have read this additional information and then declassified explicitly.

The foreign key rule addresses the deletion problem by requiring that the insert that created the problematic situation be properly vouched for. The rule recognizes that deletions expose information in the referencing table; it ensures that this is acceptable by requiring the inserter to have authority for all the tags that must be removed in order for the deleter to learn about these insertions.

The affected tags must be identified explicitly in the query. For example, to insert a Drives tuple that refers to a Cars tuple with label {alice_cars}, the process must include the clause

```
DECLASSIFYING (alice_drives, alice_cars)
```

in the insert statement and have authority for both tags. A subsequent transaction running without alice_drives in its label could discover the existence of the new Drives tuple

by attempting to delete the Cars tuple, which would fail due to the constraint. The DECLASSIFYING clause is an explicit statement that this channel is not problematic. Therefore, it supports IFDB's goal that sensitive information can only be revealed through explicit declassification.

#### 5.2.3 Generalized Constraints

Constraints that don't fall into the above categories are handled with triggers, which are stored procedures that run in response to some action, such as inserting a tuple into a particular table. Two types of triggers are available.

An ordinary trigger runs with the authority of the process that caused it to fire. Such a trigger can't leak information that the process couldn't leak, but it can only enforce the constraint with respect to the tuples the process can see. For example, when a new prescription is entered for Alice with the label {alice_medical}, a trigger checks it against her existing prescriptions for drug interactions. Since all of Alice's prescriptions have the same secrecy, this raises no information flow concerns.

Alternatively, a trigger can be defined as a stored authority closure. Such a trigger can perform stricter enforcement by using its authority. For instance, a trigger can ensure that a doctor cannot be assigned responsibility for too many patients. This trigger might "leak" which doctors are not accepting new patients; the definer of the trigger needs to decide whether this is acceptable.

When a query causes a trigger to fire at the end of a transaction, IFDB performs some extra bookkeeping to ensure that the trigger runs with the label of the query, not the commit label of the transaction. This ensures that triggers have consistent semantics regardless of whether they run immediately or are deferred until the end of the transaction. Foreign key constraints, which can be thought of as a particular kind of trigger, are handled in the same manner.

#### 5.2.4 Label Constraints

IFDB makes it possible to define constraints on labels. For example, such a constraint can specify that HIVPatients tuples should have the appropriate label, i.e., a record for Alice must have the label {alice_medical}. Label constraints can help prevent labeling errors. In addition, they can prevent polyinstantiation by augmenting a uniqueness constraint for a key to include the required label for a tuple with that key.

IFDB supports simple label constraints as a type of foreign key constraint. Other label constraints (for instance, ones that require labels of tuples table to be supersets of a given label) are handled via triggers that check the _label field.

## 6. Case Studies

This section describes our experiences porting two applications, CarTel and HotCRP, to use IFDB and PHP-IF. These applications were chosen because they have rich policies for sharing information among users. The conversion required a modest effort. We had to change 4.5% of the CarTel code base and 7% of the HotCRP code base—but in the latter case, most changes were related to converting from MySQL to PostgreSQL. IFDB prevented real privacy leaks in both systems and each had a much smaller trusted base, which was easier to reason about.

### 6.1 CarTel

CarTel [14] is a mobile sensor network that collects location data and other information from GPS-equipped cars. Users can see maps and statistics about their past drives through the CarTel website, get real-time traffic information derived from other users' drives, and compare drives with friends. CarTel is intended to protect users' privacy: The records for a car should only be accessible to its owner, but the owner can allow friends to see past drives.[1] CarTel also has sensitive data about users and cars, but for simplicity, we focus here on protecting location data.

The current CarTel implementation is a prototype produced by another research project. It uses *ad hoc* privacy controls enforced by scripts running on the web server. Each PHP script has complete access to all users' location data, and is trusted to ensure that data aren't released inappropriately. Many security bugs resulted.

We introduced two types of tags to denote the privacy concerns of location data: Alice has an alice-drives tag that covers her past drives, and an alice-location tag for her current location. The GPS coordinates and timestamps from Alice's cars are thus assigned the label {alice-drives, alice-location}, reflecting the fact that the raw location data reveals both the drive and her current location. Information about historical drives, however, is labeled with {alice-drives} so Alice can allow her friend Bob to see her drives by delegating authority for alice-drives to him.

A substantial amount of code in CarTel is involved in transforming raw location data into drives. A SQL stored procedure, driveupdate(), runs as a trigger and updates the distance traveled. Separately, a PHP function, load_drives(), interpolates the path of a drive on demand. Even though the code processes secret data, IFDB prevents it from compromising Alice's privacy. Both procedures read location data with label {alice-drives, alice-location} and write drives with label {alice-drives}. They run as authority closures with authority for the alice-location tag; however, they *cannot* declassify the alice-drives tag. Therefore, once the procedures read Alice's location data, anything they write to the database will remain contaminated, preventing it from being leaked.

The most pervasive vulnerabilities we discovered were in authentication. Twelve scripts, many of which were rarely used or intended only for testing, neglected to authenticate the user making the request. The authentication routine

[1] The full CarTel system uses a separate streaming database for real-time queries, which is not addressed in this paper. We focus on historical queries, such as comparing drives with friends.

itself also had a bug: When authentication failed, the script continued as if the user authenticated successfully, but it generated an HTTP redirect to send the user to the login page. This behavior masked the bug because a normal web browser would follow the redirect before displaying any of the sensitive information that CarTel was sending it. Another bug was related to the "friend" feature. Cartel gave each user a list of people who had designated them as friends. However, by manipulating the URL, a malicious user could see anyone's driving history.

Converting CarTel to use IFDB fixed the authentication and the authorization bugs. Scripts that didn't authenticate ran with no authority under IFDB. Furthermore, if a user attempted to coerce the site into showing the drives for a non-friend, the script would become contaminated with a tag it had no authority to declassify, and therefore it would produce no output regardless of what it read.

It's tempting to think that the kinds of bugs we found in CarTel are confined to research prototypes, but this is not the case. For example a bank and a health insurer, both Fortune 500 companies, recently exposed millions of customers' financial and medical records due to omitted authentication checks [23, 30], and the missing authorization check mirrors observed privacy holes in Facebook [4].

### 6.2 HotCRP

HotCRP [17] is a widely used conference management system. Authors submit papers, reviewers read them and enter evaluations, and the program committee (PC) produces decisions for the papers. Both papers and reviews may be anonymous. The system is intended to handle conflicts of interest so that PC members cannot see reviews for their own papers. Web users are protected from each other by logic in the application, which has access to all users' data. The privacy settings are configurable, and restrictions are implemented through hundreds of conditionals that determine what selections to add to queries and what projections to apply to the results.

We converted HotCRP to use IFDB, employing DIFC to protect data involving contact information and paper reviews and rankings. A user (say Cathy) has a tag cathy-contact protecting her tuple in the ContactInfo table. This tag is a member of the all-contacts compound tag. The PCMembers declassifying view, which has authority for all-contacts, distills the names of PC members from ContactInfo. Additionally, the program chair can record acceptance decisions, but these shouldn't be available to authors or conflicted PC members until the results are officially released; therefore, each acceptance decision is protected by a tag specific to that paper. Finally, each PaperReview tuple has a tag that only the review author and the chair are authoritative for. An authority closure running with the chair's authority later delegates the tag to eligible PC members, i.e., those with no conflicts of interest.

The resulting system blocked a previously unknown leak: the script that displayed the list of program committee members stopped working. This turned out to result from a bug that allowed any user to view the full contact information (name, address, phone number, email address, and affiliation) of all registered users.

We also reintroduced some bugs that were present in past versions of HotCRP and found they were prevented in our version. One bug allowed PC members to see decisions for their own papers prematurely by sorting papers in order of status and seeing where theirs appeared. A similar bug allowed non-PC-members to see acceptance decisions for their papers by abusing the search feature. Under our Query by Label model, the underlying database query result simply didn't return the tuples the user wasn't allowed to see.

### 6.3 Discussion

In addition to preventing bugs in CarTel and HotCRP from violating confidentiality requirements, the IFDB versions had smaller trusted bases. For example, in the original version of CarTel, a bug in any part of the application could potentially expose Alice's driving history. IFDB, however, guarantees end-to-end confidentiality for Alice's drives: Alice must trust that her location measurements are labeled properly when they enter the system, and she must trust any code that runs on behalf of her or her designated friends, but she does not need to trust any of the processing that goes on in the middle. Declassifying views and authority closures made up 380 out of 10,000 lines of code in CarTel, and 760 out of 29,000 lines in HotCRP. Additionally, each system had about 50 lines of code that were trusted to set up tags for new users and label incoming data properly.

We also found that DIFC provides a useful way to reason about security in the database. We had to categorize sensitive data (e.g., drives, locations, contact information) with tags, and determine where those tags should appear in the database. For each query, we had to consider the label the results ought to have. Furthermore, since IFDB makes declassification explicit, this drew attention to parts of the program that could potentially create vulnerabilities.

Hiding sensitive information rather than restricting access proved to be particularly useful in HotCRP, which frequently did many-way joins that included sensitive tuples, then decided later what the user ought to be able to see. We didn't want to trust the complicated application code to make the right decisions, but we also didn't want to modify the system to have one kind of query for the program chair, another for PC members, a third for authors, and a fourth for external reviewers. By using outer joins in these queries, we simply got NULLs in place of the fields that were more sensitive than the process label. Thus, we kept changes to a minimum.

### 6.4 Methodology

IFDB provides a mechanism for ensuring application security; our contribution is this mechanism. However, IFDB does *not* define what security means for a particular application. Instead, it is up to the developer of an application to decide

what this means. Our experience with using IFDB provides insight into how a developer should go about this.

The first step is to identify what information will be stored in the system, who is allowed to access it, and what kinds of computations are expected over the data. This allows the developer to identify the kinds of tags, compound tags, and principals that will be used in the application, and also the expected authority relationships: which principals should be authoritative for which tags. Typically, each kind of sensitive information will be associated with a compound tag, and the subtags will be associated with information for particular individuals. Most of these principals and tags will not be created at this point; instead they will be created when the application runs. Rather, the developer is defining a kind of *authority schema* that will be instantiated later.

Our experience indicates that it is straightforward to come up with this schema. This was true for both CarTel and HotCRP. A third example is a medical information system, where there are patient medical records, billing records, contact information, and so forth. In this system, there might be an all_patient_medical compound tag for medical records, with subtags such as alice_medical and bob_medical to identify data belonging to different patients. Each of these tags has a clear owning principal: Alice owns alice_medical, and therefore has authority to decide when to declassify for it.

The next step is to define the schema for the database tables, which is done in the usual way, and come up with a labeling strategy. In a well-designed schema, all the fields within a tuple are related, so there is an obvious tag to use: for instance, a tuple with information about Alice's latest doctor visit should have the label {alice_medical}. Larger labels are needed only if different kinds of data (for instance, both billing and medical information) are stored in the same tuple. The schema may also include label constraints: for instance, tuples for patient Alice should always have the alice_medical tag. When the application receives sensitive data, it will add the appropriate tags to its label before writing to the database.

The third step is to identify *unsafe flows* – computations where the result is less sensitive than the inputs. These flows require appropriate authority to declassify, and it is desirable to associate that authority with a minimal amount of code. Declassifying views and stored authority closures in the database, as well as authority closures in application code, provide ways to do this.

The security of the application depends on the code that runs with authority. Programmers who implement this trusted code need to understand the ramifications of doing declassifications. For example, Alice's medical bill necessarily conveys some information about Alice's medical history, and the trusted code that produces the bill is making an important policy decision. However, programmers who write the rest of the code for the application do not require the same degree of sophistication.

Our experience is that most code (including both application code and stored procedures) doesn't need to declassify or run with any special authority. This illustrates a key advantage of information flow control over access control: even code that computes on sensitive information does not have the ability to leak it.

# 7. Implementation

Our implementation has two components: the database, and the platform used by clients to interact with the database. The database stores the application data and also the *authority state*, which records the principals, tags, and delegations.

## 7.1 The Database Implementation

Our IFDB implementation is based on PostgreSQL 8.4.10, and includes about 6,300 new or modified lines of C code and 250 lines of PL/pgSQL stored procedures. Tuple labels are stored along with each tuple in a new, immutable system column called _label.

The database determines which tuples it should operate on and return for a given statement based on the label of the requesting client process. The Label Confinement Rule and the Write Rule from Section 4.2 are implemented at the layer that reads and writes tuples in tables; thus, any bugs in the higher layers that parse, optimize, and execute queries should not compromise the information flow restrictions.

The fact that PostgreSQL uses multi-version concurrency control (MVCC) made many of the changes easy to implement. In MVCC, a new version of a tuple is written every time the tuple is updated. Queries read just the relevant versions, ignoring ones that have been deleted, superceded, or not yet committed. IFDB extends the code that ignores irrelevant versions to also hide tuples according to the label of the client process. The garbage collector task that discards old versions is exempt from the information flow rules. Polyinstantiation required no special support, since the indexes that enforce uniqueness constraints already had to be prepared to deal with multiple versions.

The database provides two additional interfaces to support information flow control. The first includes the IFDB API described in Section 3, which provides SQL-callable functions for declassification, authority management, compound tags, and so forth. Syntactic extensions support pl/pgSQL authority closures and the INSERT ... DECLASSIFYING construct for foreign keys (see Section 5.2.2). The second interface is at a lower level; it allows the application platform and IFDB to communicate changes in the client process's label and authority. Changes are coalesced and transmitted lazily with the next statement or result, using a modified version of PostgreSQL's low-level client/server protocol.

Our prototype does not support indexes that would optimize queries for tuples that have specific tags in their labels. Labels are sets, typically with a small number of elements (rarely more than two), so an inverted index would be an

appropriate data structure, should the feature prove useful in the future. However, the applications we studied would not benefit from such indices. For example, CarTel looks up drives based on the car's owner rather than the label attached to the drive. Instead, labels serve as a safety net against privacy bugs: they ensure that the application's contamination reflects the drives it reads.

### 7.2 Clients

We implemented PHP-IF and Python-IF by extending PHP and Python to support DIFC. PHP-IF and Python-IF interpose on output, so programs that are too contaminated can't release information. Additionally, they have APIs similar to IFDB's to support authority closures, delegation of authority, and label changes. To provide Query by Label, they share their process label with IFDB.

The low-level details of the client/server protocol are handled by a modified version of the PostgreSQL client library, libpq. The library provides C APIs to query and change the label and principal of the process. Therefore, minimal low-level changes are required to add database support to new information flow platforms. Our PHP-IF and Python-IF application platforms, for instance, required only about 150 new lines of C code each. The remaining changes to support IFC are implemented in PHP and Python; each respective implementation is about 1100 lines of code.

The PHP implementation includes a shared memory cache of recently used principal and tag values and authority state. The cache is important because the platform frequently checks whether the current principal is allowed to release information (e.g., to the web client) given the contamination reflected in the process's label.

Command-line database clients such as psql, pg_dump, and pg_restore were also modified, mainly to provide debugging capabilities and backups that include labels.

### 7.3 Covert Channels

IFDB provides a model that handles queries, constraints, and transactions without covert channels. However, database systems are complex and their implementations can leak information, even without DIFC [12]. In line with earlier DIFC work [18, 36], the goal of our work is to develop a DIFC *model* for databases that is free of covert channels, not to thoroughly address countermeasures to channels in the implementation. This section briefly discusses some of the challenges.

A database implementation can introduce timing channels that allow a process running a query to deduce secret information from the time it takes the query to complete. Various techniques have been developed to mitigate the impact of timing channels [1, 16, 20], e.g., by quantizing response times. The IFDB prototype does not incorporate these defenses.

In addition, a database implementation can introduce allocation channels. If principals and tags were allocated in a predictable sequence, this might provide unintended information, e.g., the order in which papers were submitted in HotCRP. Therefore, IFDB uses a cryptographic pseudorandom number generator to generate new principal and tag ids.

A more subtle channel involves tuple allocation. PostgreSQL allocates storage for tuples from per-relation heap files, so the relative order of tuples within a relation is affected by the sequence of modifications to the relation. This might allow an uncontaminated process to deduce the presence of a high-labeled tuple. The channel can be avoided by ordering tuples returned to the application by any deterministic function of their values, but this is expensive. Since we expect these channels are difficult to exploit, our prototype does not reorder query results.

| Freq. | Request | Description |
|---|---|---|
| 0.50 | get_cars.php | location updates (AJAX) |
| 0.30 | cars.php | show car locations |
| 0.08 | drives.php | show drive log |
| 0.08 | drives_top.php | common driving patterns |
| 0.03 | friends.php | view and set friends |
| 0.01 | edit_account.php | edit personal info |

**Figure 3.** Distribution of HTTP requests (excluding login) generated by our CarTel web benchmark

## 8. Performance

We evaluated the performance of IFDB by comparing the end-to-end performance of our version of CarTel using IFDB and PHP-IF to that of the original version running on PostgreSQL and PHP. Our results in Section 8.2 show that the overhead is minimal, and in many cases too small to easily measure.

To gain better insight into how information flow control affects database scalability, we use a benchmark based on TPC-C to study IFDB's performance with information flow labels of various sizes. Section 8.3 presents these results.

### 8.1 Experimental Setup

We ran our benchmarks on a database server with four Xeon E7310 CPUs (16 cores), 8 GB of RAM, a RAID controller with a 256 MB battery-backed cache, and three 15,000 rpm SAS drives in a RAID 5 configuration. The server ran Linux kernel version 2.6.38. The database in each test was either IFDB or PostgreSQL 8.4.10 (from which IFDB is derived). We tuned several database parameters: checkpoint_segments=64, max_connections=1500, and the remaining settings optimized for the hardware by pg_tune 0.9.3.

Several web servers were connected to the database via a Gigabit Ethernet switch. The web servers were substantially less powerful than the database server: each had a hyper-threaded 3.06 GHz Pentium 4 CPU and 2 GB of RAM. They ran Apache 2.2.15 and either PHP-IF or PHP 5.3.10. We also used APC 3.1.3p1 to cache compiled PHP scripts.

| | Web Interactions Per Second | |
|---|---|---|
| | PostgreSQL + PHP | IFDB + PHP-IF |
| database-bound | 229.3 | 230.4 |
| web-server-bound | 132.0 | 103.5 |

**Figure 4.** Throughput of the CarTel website, with and without information flow security. The database-bound workload uses three web servers, and the web-server-bound workload uses just one.

## 8.2 Macrobenchmarks

We assessed IFDB's real-world performance by running a series of benchmarks involving CarTel. The first set of benchmarks, covered in Section 8.2.1, are read-intensive. They involve the CarTel web portal, which allows users to view location data for their cars and their friends' cars. Section 8.2.2 covers a write-intensive benchmark that measures how fast the database can process sensor measurements.

### 8.2.1 CarTel Web Portal Performance

We compared the performance of our version of the CarTel website (running on IFDB and PHP-IF) to the original version (running on PostgreSQL and PHP). The database was populated with 18 GB of real data, consisting of 177 million location measurements collected over a 27-month period.

We used a methodology based on TPC-W [34] to measure the maximum sustained throughput of the system. Simulated clients log in as a random user, make a random sequence of HTTP requests, then end their sessions. The "think time," or duration between two HTTP requests from the same client, ranges from 0 to 70 seconds, following a truncated negative exponential distribution. The length of each client session also follows a truncated negative exponential distribution, and can be up to about 60 minutes. Thus, the vast majority of think times and session durations are closer to the low ends of these ranges.

Following the initial login, simulated web clients request pages according to the distribution in Figure 3. The distribution is intended to mimic a real workload. To obtain more consistent performance, we did not generate requests for users who had more than 5 cars (one of which was a cab company). The load generator adjusted the number of clients to achieve peak throughput while keeping the 90th percentile response time under 3 seconds, which is the criterion used by TPC-W.

Figure 4 shows the maximum number of web interactions per second the web servers and database could sustain subject to the constraint on response time. With three web servers, performance was limited by the database, which was disk-bound, and we could demonstrate no statistically significant difference between IFDB and PostgreSQL over five 2-hour trials. With one web server, however, the web server's CPU was the bottleneck and throughput was 22% lower with IFDB and PHP-IF.

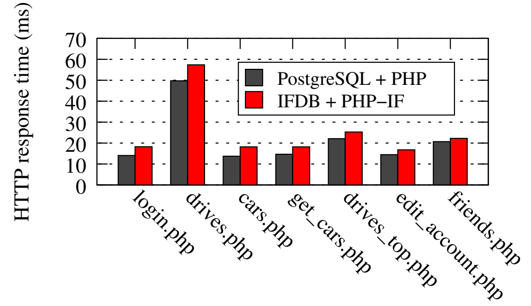


**Figure 5.** CarTel web request latency on an idle system

Figure 5 reports the HTTP request latency on an idle system, with a single client issuing requests serially. The weighted mean increase in response time with IFDB and PHP-IF was 24%. The highest absolute increase was in drives.php, which had to handle each of the user's friends. The latency difference for each script mainly reflects the additional time required to look up principal and tag ids, perform label manipulations to read sensitive data, and check that the process is allowed to release what it read. PHP-IF caches some authority information, so these checks rarely require communication with the database.

The increase in latency, and also the reduced throughput when the system is web-server bound, is due to inadequacies in our PHP-IF prototype. Much of PHP-IF is implemented in PHP itself, and performance would be greatly improved by moving this code to C. The main goal in this paper, however, is to demonstrate that DIFC on the DBMS side is practical, and our results show that DBMS performance is very good. Results from Aeolus [6] show that it is possible to achieve good performance in a DIFC application platform as well with the right optimizations, such as more sophisticated caching of authority state, and PHP-IF could take advantage of such optimizations.

### 8.2.2 Sensor Data Processing Throughput

CarTel also has a component that processes and stores GPS location measurements. For each measurement, a new tuple is inserted into the Locations table, and two triggers fire, which read from the Cars table and update the Drives and LocationsLatest tables. CarTel issues 200 inserts per transaction, partly to compensate for the lack of group commit in PostgreSQL. The web server is not involved.

In IFDB, each Locations tuple must be labeled with the appropriate user's location tag so that the platform will subsequently protect it from improper release. Additionally, the triggers run as authority closures so that they can read Cars and update Drives without contaminating the process performing the insert.

We replayed real location measurements to the database as fast as possible and measured the average throughput. PostgreSQL was able to process 2479 location measurements per second, while IFDB processed 2439. The 1.6% difference

reflects the additional bookkeeping associated with labeling the data, as well as the overhead of storing the labels themselves. The following section explores the impact of the latter source of overhead.

### 8.3 The Cost of Labels

IFDB must store a label for every tuple, and compare the labels to the process label on every read and update. This section explores those costs, independent of the other differences associated with modifying the application to support information flow control.

Labels are small in CarTel and HotCRP: there were 0 to 2 tags per tuple. Furthermore, we expect small labels in general, for two reasons. First, in a well-designed database schema, the fields in a tuple are related, so a tuple should not have many different kinds of sensitive information. Second, computations that combine many related tuples should use compound tags. For instance, the result of a statistical analysis over many CarTel users' drives can have a label containing just the all_drives tag.

Nevertheless, to gain a better understanding of the cost of labels, we evaluated IFDB's performance with tuple labels ranging from zero tags up to ten—more than we expect to see in practice. In each experiment, all of the tuples in the database had the same labels. We measured performance using DBT-2, a benchmark derived from the TPC-C [35] specification. Unlike TPC-C, we set the think time of simulated clients to zero and held the number of warehouses constant. To better capture the distinction between I/O and computational overhead, we ran the benchmark on an in-memory database with 10 warehouses and an on-disk database with 150 warehouses.

Figure 6 reports the results. The transaction rates are scaled so that performance relative to PostgreSQL is directly comparable. Within the range studied, each tag reduces throughput by about 0.6% for the in-memory workload and 1% for the on-disk workload. Since labels with one tag are the most common, we believe that 1% is a conservative estimate of the impact for real applications.

Much of the overhead comes from I/O and cache pressure. Labels in IFDB increase the size of each tuple by 4 bytes per tag, with corresponding implications for disk bandwidth and the buffer cache. (The label length is stored in a byte in the tuple header, which was previously unused for alignment reasons.) For example, Order_Line tuples, responsible for much of the I/O in TPC-C, are 89 bytes, so each tag adds 4.5% to the space consumed by the tuple, and decrease the number of tuples that can be stored per page.

## 9. Related Work

Prior work on mechanisms on information flow control has come from the database, programming language, and OS communities. IFDB incorporates ideas from all three approaches.

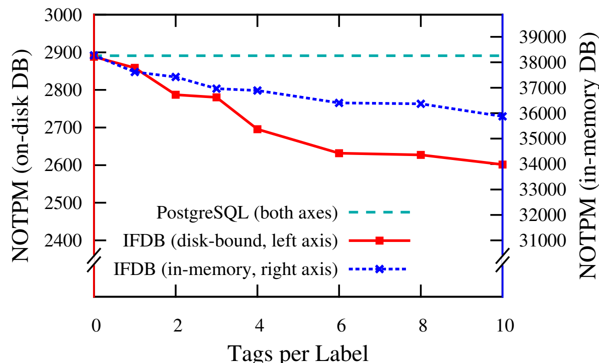


**Figure 6.** DBT-2 throughput (new-order transactions per minute)

To our knowledge there has been no work on DIFC for databases, but there has been work on information flow control. Many proposals grew from a 1982 US Air Force study; one of the most influential of these proposals came from the SeaView project [21], which developed and formally verified a query model and developed the polyinstantiation concept. IFDB draws on this work and uses similar query and data models, although IFDB's approach is simpler because it supports labeling only at tuple granularity.

SeaView was designed with military needs in mind: it assumes a central IFC policy, a set of levels (e.g., confidential, secret, top-secret), and human users who are cleared for certain levels. In contrast, the focus of IFDB is on using IFC to enable abstractions that support secure programming. As such, it includes a number of concepts not present in the SeaView model. IFDB uses decentralized IFC: it supports fine-grained tags so that principals can define policies for their own data. Additionally, IFDB provides abstractions such as declassifying views, authority closures, and reduced authority calls to limit the amount of code that must run with authority. The more sophisticated authority model in IFDB enables better solutions to problems such as covert channels due to foreign key references; in contrast, SeaView imposes the restriction that a foreign key reference must have the same label as its target.

Many MLS database systems also use implementation techniques that can't support the millions of tags potentially required by DIFC policies. SeaView partitions each table into files, one for each possible label, and queries read all of the files containing data they are allowed to see. However, that technique doesn't work if there are more than a few distinct labels. Similarly, Oracle provides multi-level security through Oracle Label Security (OLS) [15] and Virtual Private Databases, but OLS limits the number of tags (which they call *categories*) to 10,000. Sybase Secure SQL Server [33] has a maximum of 64 tags.

PostgreSQL provides security labels for tuples [26]. The database stores labels in a separate table and doesn't define

any built-in semantics for labels; instead, a loadable module must be provided to interpret labels. IFDB is based on PostgreSQL, but doesn't use the provided mechanism. Instead, we implemented our own label mechanism to achieve the required semantics and get better performance. DB2's Label-Based Access Control feature [3] is similar to PostgreSQL security labels, but DB2 provides built-in access control enforcement.

The DIFC model used in IFDB was introduced by Myers and Liskov [25]. This work has since developed into two lines of research. The first presents the model to programmers through a programming language. Languages such as JIF [24], and SIF [8] extend the type system to include labels. UrWeb [7] and a proposal by Li and Zdancewic [19] elevate queries to first-class language constructs. These extensions allow them to enforce information flow policies primarily via static analysis. The static approaches promise good performance, but they typically require programmers to learn a new language where security properties are expressed through a complex type system.

The second line of DIFC research presents the model via an operating system; it enforces information flow control at the level of operating system processes, avoiding the need for type annotations. Examples of work in this area are Asbestos [11], HiStar [36], and Flume [18]. Aeolus [6] builds on this work; it tracks labels at the level of processes but provides a higher-level interface than what an operating system would provide. All of these systems focus on file systems as the primary abstraction for persistence, although there have been some steps toward also supporting databases. For example, the HiStar paper [36] mentions a database with a limited interface, but provides no details. IFDB could be integrated with any of these systems to provide a DIFC-aware relational store for applications.

IFDB provides mechanisms to prevent bugs in applications from undermining the intended security policy but does not dictate what that policy should be. For example, it's clear that Alice ought to be able to see her own CarTel driving history, but does publishing an "anonymized" traffic report that contains her data violate her privacy? Such questions often don't have easy answers, and this paper doesn't attempt to provide one. However, recent research has developed ways to quantify privacy [10, 22, 32] and write queries that provide reasonable privacy protections [5, 13], and these results can be used to inform policy decisions in IFDB deployments.

## 10. Conclusion

This paper has described IFDB, the first system to provide security for databases through decentralized information flow control (DIFC). IFDB is intended to work with a DIFC application platform, such as PHP-IF. The system tracks sensitive information as it flows through the DBMS, and also between the application and the DBMS. The information flow label of a process becomes contaminated by what the process reads, and a process can't release information if it is too contaminated. Thus, unlike in an access control system, many computations that operate on sensitive information can run without having the authority to release it.

IFDB's Query by Label model provides a practical way for processes to control their contamination. Furthermore, the model is easy to use. It is based on familiar languages, such as SQL and PHP, and it employs simple concepts: tags, labels, and principals. IFDB also provides new database abstractions, such as declassifying views and stored authority closures, which bind the authority to remove contamination to a trusted computation. In addition, IFDB supports important database features, such as transactions and constraints, in a way that avoids information leaks via covert channels. Transactions add something new – a connection between the termination of the transaction and what happened inside it. Our transaction rules ensure that there are no leaks because of this.

We implemented IFDB by modifying PostgreSQL 8.4.10 and implemented two client-side platforms: one based on PHP and the other on Python. Our performance experiments show that our system adds minimal overhead. The case studies show it is easy to express confidentiality policies with our approach; Section 6.4 explains how to achieve similar benefits in other applications. In addition, IFDB prevented several leaks that affected the original applications, and removed most of the application code from the trusted computing base.

We are investigating three directions for future work. First, we are looking at ways to extend IFDB to enhance its usability. We are interested in adding new DIFC abstractions to the model, such as a more direct syntax for expressing common kinds of label constraints, or a special iterator where each tuple selected by a query is handled in its own context with that tuple's label. We are also interested in how to incorporate other SQL abstractions, such as sequences, into the IFDB model without introducing covert channels. Second, we would like to put the IFDB model on stronger theoretical grounds by developing proofs of noninterference. Third, we are studying improvements in the implementation, such as ways of implementing the IFDB model without placing full trust in the DBMS.

## Acknowledgments

We would like to thank Dan Ports, Nickolai Zeldovich, the referees, and our shepherd Brad Karp, for helpful comments on this paper. Sam Madden kindly provided us with the CarTel source code and data, and Eddie Kohler confirmed the bug we found in HotCRP and pointed us to some past bugs.

## References

[1] A. Askarov, D. Zhang, and A. Myers. Predictive black-box mitigation of timing channels. In *Proc. CCS*, New York, NY, 2010. ACM.

[2] D. Bell and L. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, MITRE Corp. MTR-2997, Bedford, MA, 1975.

[3] R. Bond, K. See, C. Wong, and Y.-K. Chan. *Understanding DB2 9 Security*, chapter 6. IBM Press, Indianopolis, IN, 1st edition.

[4] J. Bonneau. New Facebook photo hacks. In *Light Blue Touchpaper*. University of Cambridge Computer Laboratory, May 20, 2009. http://www.lightbluetouchpaper.org/2009/02/11/new-facebook-photo-hacks/.

[5] R. Chen, N. Mohammed, B. Fung, B. Desai, and L. Xiong. Publishing set-valued data via differential privacy. *Proc. VLDB*, 4(11), Aug. 2011.

[6] W. Cheng, D. Ports, D. Schultz, V. Popic, A. Blankstein, J. Cowling, D. Curtis, L. Shrira, and B. Liskov. Abstractions for usable information flow control in Aeolus. In *Proc. USENIX ATC*, Boston, MA, June 2012.

[7] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proc. OSDI*, Oct. 2010.

[8] S. Chong, K. Vikram, and A. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *Proc. USENIX Security*, Boston, MA, Aug. 2007.

[9] D. Denning. A lattice model of secure information flow. *Commun. ACM*, 19, May 1976.

[10] C. Dwork. Differential privacy: A survey of results. In *Proc TAMC*, Berlin, Heidelberg, 2008. Springer.

[11] 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. SOSP*, Brighton, UK, 2005. ACM.

[12] A. Futoransky, D. Saura, and A. Waissbein. The ND2DB attack: Database content extraction using timing attacks on the indexing algorithms. In *WOOT*, Boston, MA, Aug. 2007. USENIX Association.

[13] M. Hay, V. Rastogi, G. Miklau, and D. Suciu. Boosting the accuracy of differentially private histograms through consistency. *Proc. VLDB*, 3(1–2), Sept. 2010.

[14] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, A. K. Miu, E. Shih, H. Balakrishnan, and S. Madden. CarTel: A distributed mobile sensor computing system. In *SenSys*, Boulder, CO, November 2006. ACM.

[15] S. Jeloka et al. *Oracle Label Security Administrator's Guide, 11g Release 2 (11.2)*. Oracle Corporation, 2009.

[16] P. Karger and J. Wray. Storage channels in disk arm optimization. In *Proc. Security and Privacy*. IEEE, May 1991.

[17] E. Kohler. Hot crap! In *Proc. WOWCS*, Berkeley, CA, 2008. USENIX.

[18] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proc. SOSP*, New York, NY, 2007. ACM.

[19] P. Li and S. Zdancewic. Practical information-flow control in web-based information systems. In *Proc. CSFW*. IEEE, 2005.

[20] Y. Liu, D. Ghosal, F. Armknecht, A.-R. Sadeghi, S. Schulz, and S. Katzenbeisser. Hide and seek in time: Robust covert timing channels. In *Proc. ESORICS*, Berlin, Heidelberg, 2009. Springer.

[21] T. F. Lunt, D. E. Denning, R. R. Schell, M. Heckman, and W. R. Shockley. The SeaView security model. *IEEE Trans. Softw. Eng.*, 16, June 1990.

[22] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. $l$-diversity: Privacy beyond $k$-anonymity. *ACM Trans. Knowl. Discov. Data*, 1, March 2007.

[23] T. Murphy. Security glitch exposes WellPoint customers' financial, medical data. *USA Today*, June 29, 2010. URL http://www.usatoday.com/money/industries/health/2010-06-29-wellpoint-data-breach_N.htm.

[24] A. Myers. JFlow: practical mostly-static information flow control. In *POPL 1999*, San Antonio, TX, Jan. 1999. ACM.

[25] A. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. SOSP*, Saint-Malo, France, 1997. ACM.

[26] PostgreSQL Global Development Group. *PostgreSQL 9.1 Documentation*, Sept. 2011. http://www.postgresql.org/docs/9.1/static/.

[27] J. Saltzer and M. Schroeder. The protection of information in computer systems. In *Proc SOSP*, Yorktown Heights, NY, Oct. 1973.

[28] R. Sandhu and S. Jajodia. Polyinstantation for cover stories. In *Proc. ESORICS*. Springer, 1992.

[29] D. Schultz. *Decentralized Information Flow Control for Databases*. Ph.D., MIT, Cambridge, MA, Aug. 2012.

[30] N. Schwartz and E. Dash. Thieves found Citigroup site an easy entry. *The New York Times*, June 13, 2011. URL https://www.nytimes.com/2011/06/14/technology/14security.html.

[31] K. Smith and M. Winslett. Entity modeling in the MLS relational model. In *Proc. VLDB*, San Francisco, CA, 1992. Morgan Kaufmann.

[32] L. Sweeney. $k$-anonymity: A model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10, Oct. 2002.

[33] Sybase, Inc. *Final Evaluation Report: SQL Server Version 11.0.6 and Secure SQL Server Version 11.0.6*, chapter 5: Security Architecture. National Computer Security Center, Ft. Meade, MD, Mar. 1997.

[34] *TPC Benchmark W (Web Commerce) Specification*. Transaction Processing Performance Council, San Jose, CA, 1.8 edition, February 2000.

[35] *TPC Benchmark C Specification*. Transaction Processing Performance Council, San Jose, CA, 5.11 edition, February 2010.

[36] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. OSDI*, Berkeley, CA, 2006. USENIX.