Fine-Grained Disclosure Control for App Ecosystems

Gabriel M. Bender, Lucja Kot, and Johannes Gehrke
Cornell University
Ithaca, NY 14853, USA
{gbender, lucja, johannes}@cs.cornell.edu

Christoph Koch
EPFL
CH-1015 Lausanne, Switzerland
christoph.koch@epfl.ch

ABSTRACT

The modern computing landscape contains an increasing number of *app ecosystems*, where users store personal data on platforms such as Facebook or smartphones. APIs enable third-party applications (apps) to utilize that data. A key concern associated with app ecosystems is the confidentiality of user data.

In this paper, we develop a new model of disclosure in app ecosystems. In contrast with previous solutions, our model is data-derived and semantically meaningful. Information disclosure is modeled in terms of a set of distinguished *security views*. Each query is *labeled* with the precise set of security views that is needed to answer it, and these labels drive policy decisions.

We explain how our disclosure model can be used in practice and provide algorithms for labeling conjunctive queries for the case of single-atom security views. We show that our approach is useful by demonstrating the scalability of our algorithms and by applying it to the real-world disclosure control system used by Facebook.

Categories and Subject Descriptors

H.0 [**Information Systems**]: General

Keywords

database security; view rewriting; app ecosystems

1. INTRODUCTION

The rise of Web 2.0 has caused a tremendous interest in sharing information online and a correspondingly large concern about privacy. Personal data is increasingly published, archived, re-shared and re-sold to third parties in complex software ecosystems [4]. These systems are architecturally diverse, ranging from centralized settings like a mobile device running apps, to hybrid device/cloud solutions such as Box's OneCloud platform [1], to Facebook apps that make use of personal-data APIs but run on their own servers. However, at the core of each system there is a *database* containing private data whose disclosure and propagation must be controlled. High-profile incidents of unintended disclosure make the news on a regular basis and have serious consequences for end users and other parties such as the owners of the platform and app developers [5].

We argue that each app ecosystem needs a principled, system-wide solution for privacy. There are two fundamental components to any such solution. The first is a formal model of information disclosure that allows the formulation of precise security policies. The second is an end-to-end mechanism for tracking the flow of information through the system and ensuring that the security policies are enforced. Although tracking information flow is challenging, there is substantial work on program analysis and language-based information flow [28] that can be usefully applied to app ecosystems and has already been implemented on mobile devices in PiOS [13] and TaintDroid [14]. This paper addresses the first issue and presents a novel model of information disclosure for app ecosystems.

Of course, there has been a substantial amount of research on database access control, and there are even systems such as IFDB [29] that control both disclosure and information flow. Also, in practice, app ecosystems do have a simple way of controlling disclosure: they allow users to set data permissions. Each permission conceptually regulates access to a specific *view* of the data.

However, existing approaches to disclosure control have some major disadvantages. There is evidence that both users and developers misunderstand and misuse app permissions [17]. As new data is added to the database and old data is put to new and unexpected uses, the permissions structure can get out of date and inconsistencies occur. For example, the Facebook permission named `user_likes` confusingly gives apps access to both a user's "Liked" pages and the languages the user speaks.

The source of these problems is that existing models of disclosure are not *data-derived*. In a data-derived disclosure model, the disclosure associated with any view over the database is a mathematical function of the data needed to compute the view. This is very different from ad-hoc hand-generated disclosure descriptions such as the Facebook `user_likes` permission; a data-derived approach would avoid the semantic drift mentioned above. A further advantage of a data-derived approach is that the information disclosed by a view can be computed *automatically* from the view definition, which reduces the burden on humans and makes the process less error-prone.

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

Meetings

Time	Person
9	Jim
10	Cathy
12	Bob

Contacts

Person	Email	Position
Jim	jim@e.com	Manager
Cathy	cathy@e.com	Intern
Bob	bob@e.com	Consultant

(a)

$$V_1(x, y) :- \texttt{Meetings}(x, y)$$
$$V_2(x) :- \texttt{Meetings}(x, y)$$
$$V_3(x, y, z) :- \texttt{Contacts}(x, y, z)$$

(b)

$$Q_1(x) :- \texttt{Meetings}(x, \texttt{'Cathy'})$$
$$Q_2(x) :- \texttt{Meetings}(x, y) \wedge \texttt{Contacts}(y, w, \texttt{'Intern'})$$

(c)

Figure 1: (a) Dataset (b) Security views (c) Queries

However, being data-derived is not enough; for example, a definition of disclosure in terms of the number of bits that are being disclosed (in the spirit of Differential Privacy [12]) is unlikely to be meaningful to a user. Thus we need a notion of disclosure that is not just data-derived, but also *semantically meaningful*, which means that disclosure maps to a concept that the user can intuitively understand.

The third requirement for a disclosure model is that it should be *expressive*. While the security policies for games and social apps are often simple, the corporate world is also becoming increasingly dependent on app ecosystems through BYOD (Bring Your Own Device) solutions [1]. Mobile apps are also used in the military [3]; both of these use cases demand significantly more complex security policies.

1.1 Our solution

Our solution for defining a data-derived, semantically meaningful and expressive notion of disclosure is based on the idea of associating levels of disclosure with views over the database. Some views disclose more information than others; this induces a hierarchy. For example, consider a user Alice whose calendar and contacts data is shown in Figure 1 (a). The view V_1 contains information from the full `Meetings` table, while V_2 displays only the time slots of appointments. Clearly V_2 reveals less information than V_1. [1]

Although apps may ask arbitrary queries on the data, Alice can formulate a policy based on a small set of *security views* such as those in Figure 1 (b). For example, she may specify that she is happy to disclose V_2 but not V_1; therefore, any query that can be answered using only information in V_2 is permitted but queries requiring more information are forbidden. To enforce this policy, every query Q is *automatically* associated ("labeled") with the set of security views that is required to answer Q but reveals as little information as possible beyond that. For example, the label of Q_1 in Figure 1 (c) is $\{V_1\}$ and the label of Q_2 is $\{V_1, V_3\}$. Policies are defined in terms of labels; Alice can specify that any query whose label is just $\{V_2\}$ can be answered, but queries with labels that are "above" (more informative than) V_2 should be rejected. Both Q_1 and Q_2 would be rejected under such a policy.

[1]Many app ecosystems, including the iOS and Android mobile platforms, contain such data, and this type of data is typically considered sensitive by users [13, 16]. In fact, LinkedIn recently came under fire for writing an iOS app that sent sensitive information extracted from users' calendars back to LinkedIn's servers [25].

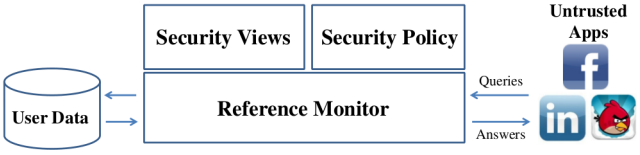

Figure 2: Disclosure control system model

Figure 2 illustrates the full workflow. The user, perhaps with assistance by the platform developer and/or third party privacy watchdog groups, creates a base set of security views. Upon this set of views, the user defines a security policy that specifies what level of disclosure is permitted; again, other parties could help by pre-defining sensible policies which users could adjust as desired. The disclosure restriction is enforced by a *reference monitor* – a component that automatically computes the labels for all incoming queries, and accepts or rejects queries to ensure that the security policy is never violated. Figure 2 is a conceptual diagram rather than an architectural specification. In practice, the reference monitor could be an independent system component or a part of the DBMS or embedded within the untrusted app.

1.2 Our contributions

We make the following contributions:

- We develop a novel model of disclosure for app ecosystems that satisfies the desiderata we identified: (1) Our model is data-derived because the labeler associates each query directly with the information required to answer it, (2) it is semantically meaningful because each query is labeled with a set of views that concisely characterizes the information that it discloses (Sections 2 and 3), and (3) it is expressive enough to support sophisticated security policies that are challenging to capture with conventional view-based security mechanisms. (Section 3.4)
- We develop practical algorithms for disclosure labeling and policy enforcement (Sections 4, 5 and 6).
- We apply our framework to practical disclosure control scenarios. As a case study, we review Facebook's handcrafted permissions labeling of FQL and Graph API queries and discover multiple inconsistencies and problems with the documentation. We also show that our algorithms scale well on realistic workloads (Section 7).

2. PROBLEM DEFINITION

In this section we begin formalizing *disclosure labeling*, which is central to our work. We explain how it provides a foundation for disclosure control and we compare this approach with existing solutions.

2.1 Disclosure labeling

The goal of disclosure labeling is to determine what information about the underlying database is disclosed by answering an arbitrary set of queries $\mathcal{Q}$. Labeling makes use of a set $\mathcal{S}$ of *security views* which reveal known and semantically meaningful types of information about the database. A *disclosure labeler* is a function which relates the information revealed by $\mathcal{Q}$ to the information revealed by the security views in $\mathcal{S}$. Specifically, it identifies a subset of $\mathcal{S}$ which

is sufficient to answer all the queries in $\mathcal{Q}$, but otherwise discloses as little additional information as possible.

Disclosure labelers provide a formal foundation for the disclosure control in app ecosystems which is our ultimate goal. In addition, they highlight previously unexplored connections between view-based security and order theory.

Defining disclosure labelers and providing labeling algorithms are both challenging. A precise definition of disclosure labeling requires us to formalize the concept of disclosing "as little information as possible" about the database. Existing theoretical work [20] addresses this to an extent, but has limited practical applicability.

In the remainder of this Section, we situate disclosure labeling within a broader problem space; we present our solution to the above challenges starting in Section 3.

2.2 Related problems and approaches

Disclosure labeling is related to equivalent query rewriting, which takes as input a query and a set of views and tries to find an equivalent rewriting of the query in terms of the views. This is a well-studied problem [9, 10, 18, 26]. There are also algorithms to find rewritings that minimize some real-valued cost metric, such as the number of views used [7]. The main difference in our case is that disclosure labeling additionally requires the set of views $\mathcal{S}'$ to disclose a minimal amount of information about the dataset.

As explained at a high level in the introduction and formally in Section 3.4, disclosure labelers can be used to enforce policies relating to disclosure control. There are existing related view-based security solutions. The SQL `GRANT` and `REVOKE` keywords are used for coarse-grained access control, and extensions to these exist [27]. There are also similarities with existing permissions systems such as Facebook's. Facebook exposes its data through the Graph API and FQL [2]. For each API and each query, the documentation specifies which permissions an app must hold (i.e., which security views it must be able to access) to receive an answer. This association between queries and permissions is a simple labeling generated by Facebook's engineers; to our knowledge the labeling is created by manual inspection of the queries.

Our approach to disclosure control using disclosure labels generalizes the solutions above and has significant additional benefits. First, our framework allows a system to keep track of *cumulative* information disclosure across multiple queries. We can determine whether each new query would push the total amount of information disclosed beyond the user's desired threshold. Second, our abstractions allow the natural formulation of complex security policies. For example, suppose Alice, whose data is shown in Figure 1 is willing to disclose either her meetings or her list of contacts, but not both. We can enforce this policy by only allowing sequences of queries whose labels are strictly below $\{V_2, V_3\}$. Third, our formal approach allows us to reason precisely about the information disclosed by the security views to identify overlap, redundancies, and inconsistencies in the policy. If the original policy and/or labeling is hand-crafted, the ability to identify such problems is especially important.

Disclosure labeling has further applications. We can use labeling to discover precisely how much information is disclosed by a given query which is useful in declassification [28]. Labeling also makes it possible to detect overprivileged applications that request access to more permissions than they need due to developer error.

2.3 Notation and terminology

We close this section by introducing a few additional pieces of terminology and notation. Although Sections 3 and 4 do not assume any particular query language, we will later restrict our attention to conjunctive queries over the schema of a fixed database D. A conjunctive query has the form

$$H :- B$$

where H is a relational atom and B a conjunction of relational atoms over database relations. H and B are the *head* and *body* of the query, respectively. Each atom may contain constants and variables. Any variables that appear in H must also appear in B. Letters x, y, z etc. indicate variables and letters a, b, c etc. indicate constants. A *distinguished* variable is one that appears in the head of the query, and an *existential* variable is one that appears only in the body. We say that two queries are *equivalent* if they return the same answer on every dataset.

Section 3 formalizes disclosure in terms of preorders and lattices. Given a set $\mathcal{C}$, a binary relation is a subset of $\mathcal{C} \times \mathcal{C}$. A relation $\sim$ is *reflexive* if $c \sim c$ for all $c \in \mathcal{C}$; it is *symmetric* if $c \sim c'$ if and only if $c' \sim c$; it is *antisymmetric* if $c \sim c'$ and $c' \sim c$ together imply $c = c'$; and it is *transitive* if $c \sim c'$ and $c' \sim c''$ together imply $c \sim c''$. A *preorder* is a binary relation that is reflexive and transitive. An *equivalence relation* is a preorder that is also symmetric. A *partial order* is a preorder that is also antisymmetric. If $\preceq$ is a partial order, then $\mathcal{C}$ forms a *lattice* under $\preceq$ if every pair of elements from $\mathcal{C}$ has both a least upper bound (LUB) and greatest lower bound (GLB). A *bounded lattice* contains a least element $\bot$ and a greatest element $\top$; all lattices we consider are bounded.

3. DISCLOSURE LABELING

This section introduces a formal framework for measuring and controlling disclosure that draws on fundamental connections between view-based security and order theory. *Disclosure orders* (Section 3.1) and *disclosure lattices* (Section 3.2) allow us to reason about the amounts of information disclosed by different set of views. We formally define *disclosure labelers* in Section 3.3. It turns out not every set of security views is suitable for creating a disclosure labeler; we give conditions that guarantee the existence and uniqueness of disclosure labelers. In Section 3.4, we explain the conceptual end-to-end setup for the use of disclosure labeling in disclosure control, including a formal definition of a security policy. Our notation is summarized in Table 1.

3.1 Disclosure orders

We now define *disclosure orders*, which formalize the notion that some sets of views disclose more information than others. Assume all views are drawn from a finite universe $\mathcal{U}$. A disclosure order $\preceq$ ranks the relative information revealed by different sets of views. Roughly speaking, we say that $\mathcal{W}_1 \preceq \mathcal{W}_2$ precisely when all the information revealed by $\mathcal{W}_1$ is also revealed by $\mathcal{W}_2$.

A natural candidate for such an order is based on *view determinacy* [23]. Under this order, $\mathcal{W}_1 \preceq \mathcal{W}_2$ precisely when the answers to all the views in $\mathcal{W}_1$ are uniquely determined by the answers to the views in $\mathcal{W}_2$. Unfortunately, checking this criterion is highly intractable for many classes of queries. *Equivalent view rewriting* provides a conservative approximation to the determinacy ordering in which

Notation	Description
$\mathcal{U}$	universe of all possible queries
$\mathcal{S}$	set of security views, $\mathcal{S} \subseteq \mathcal{U}$
$\mathcal{W}, \mathcal{W}_1, \mathcal{W}_2$	sets of views; $\mathcal{W}, \mathcal{W}_1, \mathcal{W}_2 \subseteq \mathcal{U}$
V, V_1, V_2	views; $V, V_1, V_2 \in \mathcal{U}$
Q, Q_1, Q_2	queries (to be labeled); $Q, Q_1, Q_2 \in \mathcal{U}$
$\preceq$	disclosure order
$(\Downarrow \mathcal{W})$	set of all views below $\mathcal{W}$ under $\preceq$
$\mathcal{I}$	elements of the disclosure lattice over $\mathcal{U}$ induced by $\preceq$
$\wp(\mathcal{U})$	power set of $\mathcal{U}$, i.e. collection of all subsets of $\mathcal{U}$
$\mathcal{F}$	set of disclosure labels, where each label is a set of views; $\mathcal{F} \subseteq \wp(\mathcal{U})$
ℓ	a disclosure labeler
$\ell(\mathcal{I})$	all elements of the lattice of disclosure labels for ℓ
$\mathcal{P}$	a security policy; can be represented as a subset of $\ell(\mathcal{I})$

Table 1: Notation summary

$\mathcal{W}_1 \preceq \mathcal{W}_2$ precisely when, for each view $W \in \mathcal{W}_1$, there exists an equivalent rewriting of W in terms of the views in $\mathcal{W}_2$. In contrast to determinacy, equivalent view rewriting is known to be tractable for many classes of queries [21].

The choice of disclosure order will depend on multiple factors: required throughput, sensitivity to false negatives, and the complexity of the query language under consideration. Rather than restrict ourselves to one of the orders defined above, we develop a more general framework that is applicable to *any* preorder which satisfies two basic properties. The first states that adding new elements to a set of views can only increase the amount of information that it reveals about the database. The second allows us to derive meaningful upper bounds on information disclosure even for adversaries who combine information from multiple sources.

Definition 3.1. *A **disclosure order** is a preorder on $\wp(\mathcal{U})$ that satisfies the following two properties:*

(a) If $\mathcal{W}_1 \subseteq \mathcal{W}_2$ then $\mathcal{W}_1 \preceq \mathcal{W}_2$.

(b) If $\phi \subseteq \wp(\mathcal{U})$ and $\mathcal{W} \preceq \mathcal{W}_0$ for all $\mathcal{W} \in \phi$ then $\bigcup \phi \preceq \mathcal{W}_0$.

Both orders mentioned above are disclosure orders, but others exist too. One example is the usual set order, where $\mathcal{W}_1 \preceq \mathcal{W}_2$ precisely when $\mathcal{W}_1 \subseteq \mathcal{W}_2$.

Disclosure orders need not be partial orders, as they are in general *not* antisymmetric. For example, consider the following two views on `Meetings`, abbreviated as `M`.

$$V_1(x, y) :- \texttt{M}(x, y) \qquad V_1'(y, x) :- \texttt{M}(x, y)$$

V_1 and V_1' each disclose all of `M`, so $\{V_1\} \preceq \{V_1'\}$ and $\{V_1'\} \preceq \{V_1\}$ under determinacy and equivalent view rewriting, but the two sets are clearly not equal.

Despite being unequal, $\{V_1\}$ and $\{V_1'\}$ reveal equivalent information about `M`, since each set can be computed from the other. More generally, the relation defined by $\mathcal{W}_1 \equiv \mathcal{W}_2$ if $\mathcal{W}_1 \preceq \mathcal{W}_2$ and $\mathcal{W}_2 \preceq \mathcal{W}_1$ is an equivalence relation.

3.2 Disclosure lattices

Before we move to labeling, there are two more foundational questions to address. First, given two sets of views $\mathcal{W}_1$ and $\mathcal{W}_2$, what information is disclosed to someone who knows both of them beyond what could have been inferred from just one of the two sets? This is the classical problem of *information combination*. Second, what common information, or *overlap*, is there in the information revealed by the two sets of views?

One might be tempted to use the union and intersection of $\mathcal{W}_1$ and $\mathcal{W}_2$ to answer the above questions. However, intersection does not work as a measure of overlap. To see this, consider the views V_2 and V_4 in Figure 3. The two sets $\{V_2\}$ and $\{V_4\}$ have an empty intersection, but they do have some overlap. Notably, given the answer to either V_2 or V_4, it is possible to deduce whether `Meetings` is nonempty, i.e. to answer the query V_5.

To define combination and overlap, we introduce a new operator $(\Downarrow \mathcal{W})$ that returns all the views in $\mathcal{U}$ whose answers can be inferred by observing a set of views $\mathcal{W}$. This more accurately represents all the information disclosed by $\mathcal{W}$.

Definition 3.2. *Let $\mathcal{W} \subseteq \mathcal{U}$. Then*

$$(\Downarrow \mathcal{W}) = \{V \in \mathcal{U} : \{V\} \preceq \mathcal{W}\}$$

If $\mathcal{W}_1 \preceq \mathcal{W}_2$ then every view whose answer can be computed from $\mathcal{W}_1$ can also be computed from $\mathcal{W}_2$. Furthermore, $\mathcal{W}_1 \preceq \mathcal{W}_2$ if and only if $(\Downarrow \mathcal{W}_1) \subseteq (\Downarrow \mathcal{W}_2)$.

The $\Downarrow$ operator now allows us to formalize information combination and overlap. Given $\mathcal{W}_1$ and $\mathcal{W}_2$, the information that can be derived from both $\mathcal{W}_1$ and $\mathcal{W}_2$ taken together is $\Downarrow (\mathcal{W}_1 \cup \mathcal{W}_2)$, and the information overlap of $\mathcal{W}_1$ and $\mathcal{W}_2$ is $(\Downarrow \mathcal{W}_1) \cap (\Downarrow \mathcal{W}_2)$. We can use these functions to construct a lattice structure that precisely captures the information disclosed by each subset of $\mathcal{U}$.

Theorem 3.3. *Let $\mathcal{U}$ be a set of views, and let $\preceq$ be a disclosure order for $\mathcal{U}$. Define $\mathcal{I} = \{(\Downarrow \mathcal{W}) : \mathcal{W} \subseteq \mathcal{U}\}$. Then $\mathcal{I}$ is a lattice under the subset ordering with details as follows:*

(a) LUB: $(\Downarrow \mathcal{W}_1) \sqcup (\Downarrow \mathcal{W}_2) = \Downarrow (\mathcal{W}_1 \cup \mathcal{W}_2)$.

(b) GLB: $(\Downarrow \mathcal{W}_1) \sqcap (\Downarrow \mathcal{W}_2) = (\Downarrow \mathcal{W}_1) \cap (\Downarrow \mathcal{W}_2)$.

(c) Top element $\top = (\Downarrow \mathcal{U}) = \mathcal{U}$, bottom element $\bot = (\Downarrow \emptyset)$.

We call this lattice the *disclosure lattice over $\mathcal{U}$*. It is a strict generalization of the Lattice of Information [20].

As an example, consider the `Meetings` relation from Figure 1 and suppose our universe $\mathcal{U}$ consists of the four views in Figure 3. Letting $\preceq$ be the equivalent view rewriting ordering, the disclosure lattice for this $\mathcal{U}$ is shown in Figure 3. The GLB of $\Downarrow \{V_2\}$ and $\Downarrow \{V_4\}$ is $\Downarrow \{V_5\}$. Their LUB is not $\Downarrow \{V_1\}$ but another properly lower element, accurately reflecting the fact that it is impossible to reconstitute the `Meetings` relation from the projections on its two attributes.

3.3 Disclosure labelers

In this section, we define disclosure labelers and explain under what conditions they exist. We begin with a set of security views $\mathcal{S}$, each of which reveals a known type of information about the dataset. A labeler ℓ is a function that expresses the information revealed by an unknown set of queries $\mathcal{Q}$ in terms of the information revealed by a subset $\mathcal{S}' \subseteq \mathcal{S}$.

It would seem that ℓ should map subsets of $\mathcal{U}$ to subsets of $\mathcal{S}$. For technical reasons, we permit the labeler's output to range over elements of an arbitrary set $\mathcal{F}$, even if $\mathcal{F}$ is not

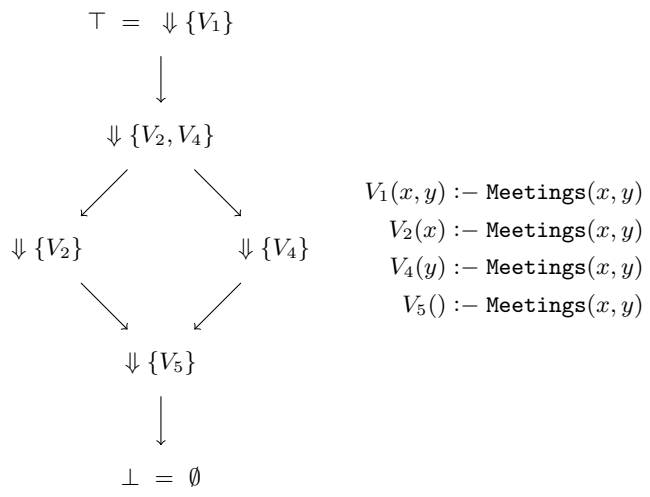

Figure 3: Disclosure lattice (left) and corresponding views (right).

a power set. Furthermore, the output of ℓ does not need to be an element of $\mathcal{F}$ so long as it is *equivalent* to an element of $\mathcal{F}$.

We place three additional restrictions on ℓ in order to ensure that the labels it finds are semantically meaningful. First, if $\mathcal{W} \in \mathcal{F}$ then $\ell(\mathcal{W})$ should reveal the same information as $\mathcal{W}$. This ensures that the labeler behaves correctly for "easy" inputs. It also means that the elements in $\mathcal{F}$ are the *fixpoints* of ℓ, which motivates our choice of notation $\mathcal{F}$. Second, the labeler should never *underestimate* the amount of information disclosed by a set of queries. And third, ℓ should be monotonic – if one set of views reveals less than another then the label of the first set of views should be below the label of the second.

Definition 3.4. *Let $\mathcal{F}$ be a subset of $\wp(\mathcal{U})$, the power set of $\mathcal{U}$, and assume $\preceq$ is a disclosure order. A* ***disclosure labeler*** *is a map $\ell : \wp(\mathcal{U}) \to \wp(\mathcal{U})$ such that*

(a) If $\mathcal{W}_1 \subseteq \mathcal{U}$ then $\ell(\mathcal{W}_1) \equiv \mathcal{W}_2$ for some $\mathcal{W}_2 \in \mathcal{F}$.

(b) If $\mathcal{W} \in \mathcal{F}$ then $\ell(\mathcal{W}) \equiv \mathcal{W}$.

(c) If $\mathcal{W} \subseteq \mathcal{U}$ then $\mathcal{W} \preceq \ell(\mathcal{W})$.

(d) If $\mathcal{W}_1, \mathcal{W}_2 \subseteq \mathcal{U}$ and $\mathcal{W}_1 \preceq \mathcal{W}_2$ then $\ell(\mathcal{W}_1) \preceq \ell(\mathcal{W}_2)$.

We call $\mathcal{F}$ the set of *disclosure labels* for ℓ.

The axioms defined above mirror those in the definition of an order-theoretic *closure operator* [11]. In fact, if $\mathcal{I}$ is the disclosure lattice of $\mathcal{U}$ then the operator that maps every $X \in \mathcal{I}$ to $(\Downarrow \ell(X))$ is a closure operator on $\mathcal{I}$.

Consider now the practical scenario where we start with a set of security views $\mathcal{S}$ and would like to define a labeler where $\mathcal{F} = \wp(\mathcal{S})$. Unfortunately, we cannot always do this, as the following example shows.

Example 3.5. Consider V_2 and V_4 from Figure 3, and suppose we want to label queries with $\wp(\{V_2, V_4\})$. This gives the following $\mathcal{F} = \{\emptyset, \{V_2\}, \{V_4\}, \{V_2, V_4\}, \top\}$.[2] Suppose that, as in previous examples, $\mathcal{U}$ consists of the four views in Figure 3. It turns out that there is no labeler for $\mathcal{U}$ using $\mathcal{F}$ as the set of disclosure labels. To see this, suppose a labeler does exist and consider what $\ell(V_5)$ might be. Since $\{V_5\} \preceq \{V_2\}$, we know that $\ell(\{V_5\}) \preceq \ell(\{V_2\}) \equiv \{V_2\}$. Similarly, since $\{V_5\} \preceq \{V_4\}$, we know that $\ell(\{V_5\}) \preceq \ell(\{V_4\}) \equiv \{V_4\}$. Since $\ell(\{V_5\}) \in \mathcal{F}$, we are forced to conclude that $\ell(\{V_5\}) = \emptyset$. However, $\{V_5\} \not\preceq \emptyset$, violating condition (c) of Definition 3.4.

[2]The disclosure labeler axioms imply that $\mathcal{F}$ contains $\top$.

The conditions in Definition 3.4 are much stronger than they first appear. It is not always possible to define a disclosure labeler for a given $\mathcal{F}$; however, when a disclosure labeler *does* exist, it is unique up to equivalence. To formalize and prove this, it is necessary to understand how labelers interact with the disclosure lattice defined in Section 3.2.

If we apply a disclosure labeler ℓ to each element of $\mathcal{I}$ (the disclosure lattice of $\mathcal{U}$), we obtain a new lattice $\ell(\mathcal{I})$.

Theorem 3.6 (Labeling on the disclosure lattice). *Define $\ell(\mathcal{I}) = \{\Downarrow \ell(\mathcal{W}) : \mathcal{W} \in \mathcal{I}\}$. Then $\ell(\mathcal{I})$ is a lattice under the subset ordering, with GLB and LUB as follows.*

(a) GLB: $(\Downarrow \mathcal{W}_1) \sqcap_\ell (\Downarrow \mathcal{W}_2) = (\Downarrow \mathcal{W}_1) \sqcap (\Downarrow \mathcal{W}_2)$.

(b) LUB: $(\Downarrow \mathcal{W}_1) \sqcup_\ell (\Downarrow \mathcal{W}_2) = \Downarrow \ell((\Downarrow \mathcal{W}_1) \sqcup (\Downarrow \mathcal{W}_2))$.

We call this lattice the *lattice of disclosure labels*. Roughly speaking, each element in this lattice corresponds to the information revealed by an element of $\mathcal{F}$. More formally, it is possible to show that $\ell(\mathcal{I}) = \{\Downarrow \mathcal{W} : \mathcal{W} \in \mathcal{F}\}$. For arbitrary $\mathcal{W}_1, \mathcal{W}_2 \in \mathcal{F}$ this lattice is guaranteed to contain the GLB $(\Downarrow \mathcal{W}_1) \sqcap (\Downarrow \mathcal{W}_2)$. It will also contain a suitable $\Downarrow \mathcal{W}_3$, for $\mathcal{W}_3 \in \mathcal{F}$, which can serve as a LUB. However, this new LUB may be higher in the original disclosure lattice than $(\Downarrow \mathcal{W}_1) \sqcup (\Downarrow \mathcal{W}_2)$.

Theorem 3.6 provides the insight to characterize when a set $\mathcal{F}$ can be used to formulate a disclosure labeler.

Theorem 3.7 (Labeler Existence). *Let $\mathcal{F} \subseteq \wp(\mathcal{U})$, and assume $\preceq$ is a disclosure order. There exists a disclosure labeler ℓ with domain $\mathcal{U}$ and image $\mathcal{F}$ precisely when $K = \{\Downarrow \mathcal{W} : \mathcal{W} \in \mathcal{F}\}$ has the following properties:*

(a) For each $X_1, X_2 \in K$, we have $X_1 \sqcap X_2 \in K$, and

(b) K contains $\mathcal{U}$.

Intuitively, $\mathcal{F}$ can be used to formulate a disclosure labeler if the corresponding set it induces in the disclosure lattice is both closed under GLB and contains $\top$. If a labeler does exist, it is unique up to equivalence. This allows us to formulate the following definition:

Definition 3.8 (Inducing labelers). *Let $\mathcal{F} \subseteq \wp(\mathcal{U})$. We say $\mathcal{F}$* induces a disclosure labeler *on $\mathcal{U}$ if it satisfies the condition in Theorem 3.7. We call the labeler ℓ from Theorem 3.7 the labeler* induced by *$\mathcal{F}$.*

If $\mathcal{F}$ induces a disclosure labeler, the following algorithm is a naïve but correct implementation of that labeler. It assumes that no two distinct elements of $\mathcal{F}$ are equivalent. The algorithm sorts the elements of $\mathcal{F}$ in order of increasing disclosure (Lines 2–3) and finds the first element in the new order that reveals at least as much information as $\mathcal{W}$ (Lines 4–8).

```
1: procedure NaïveLabel(F, W)
2:     Let F[1], ..., F[n] be the elements of F.
3:     Sort F so that if F[i] ⪯ F[j] then i ≤ j.
4:     for i ← 1, 2, ..., n do
```

```
5:        if W ⪯ F[i] then
6:            return F[i]
7:        end if
8:    end for
9:    return ⊤
10: end procedure
```

This completes our presentation of disclosure labelers. Note that the output of a labeler satisfies our desiderata from Section 1: It is a data-derived measure of disclosure because it is a mathematical function of the data needed to answer the queries. The output is also semantically meaningful as it expresses disclosure by relating it to security views that the user understands, and the model is expressive due to the ability to choose a large and complex $\mathcal{F}$.

3.4 From labelers to security policies

We have defined *disclosure labelers* which allow us to restate the information revealed by an unknown set of queries in terms of the information revealed by a much smaller, known, set of security views. Labelers are useful because they allow the formulation of semantically meaningful security policies; we explain this now in more detail.

As we saw, the information associated with the disclosure labels for a given labeler can be represented as a lattice $\ell(\mathcal{I})$. Conceptually, a security policy is a cut in this lattice: a set of queries whose label is below the cut can be answered, but a set of queries whose label falls above the cut cannot. We can formally represent the security policy by the set of elements in the lattice that are below the desired cut.

Definition 3.9 (Security policy). *A security policy $\mathcal{P}$ for labeler ℓ is a subset of the elements in the lattice of disclosure labels for ℓ.*

For example, let $\mathcal{U}$ contain the four views in Figure 3, and let ℓ be a trivial disclosure labeler that maps every subset of $\mathcal{U}$ to itself. $\mathcal{P} = \{\bot, \Downarrow\{V_5\}, \Downarrow\{V_2\}, \Downarrow\{V_4\}\}$ represents a policy in which either the first or the second attribute of `Meetings` may be disclosed, but not both. This is an example of a *Chinese Wall* policy [8] and shows that our framework is powerful enough to express fairly complex policies cleanly.

An important restriction is that policies must be *internally consistent* in the sense that if $\mathcal{W} \preceq \mathcal{W}'$ and $\Downarrow \mathcal{W}' \in \mathcal{P}$ then $\Downarrow \mathcal{W} \in \mathcal{P}$. In our running example, a principal who can view the entirety of the `Meetings` relation should also be permitted to view the projections on each attribute.

In practice, we assume queries arrive in the system one at a time. A reference monitor is an algorithm that inspects each query and accepts or rejects it to ensure the policy is never violated. A simple algorithm for enforcing a security policy $\mathcal{P}$ while answering a set of queries $\mathcal{Q}$ is given below.

```
1: L_cum ← ∅
2: for Q ∈ Q do
3:     L_new ← ℓ(Q ∪ L_cum)
4:     if (⇓ L_new) ∈ P then
5:         answer Q
6:         L_cum ← L_new
7:     else
8:         refuse Q
9:     end if
10: end for
```

The algorithm processes incoming queries in $\mathcal{Q}$ one at a time (Line 2). It first computes the total information disclosed if the query would be answered (Line 3). If such disclosure is permitted by the policy (Line 4), the query is answered and the cumulative disclosure $\mathcal{L}_{cum}$ is updated (Lines 5 and 6). This completes our presentation of disclosure labelers. We now show how they can be made practical.

$$V_3(x,y,z) :- \texttt{C}(x,y,z) \qquad V_9(x) :- \texttt{C}(x,y,z)$$
$$V_6(x,y) :- \texttt{C}(x,y,z) \qquad V_{10}(y) :- \texttt{C}(x,y,z)$$
$$V_7(x,z) :- \texttt{C}(x,y,z) \qquad V_{11}(z) :- \texttt{C}(x,y,z)$$
$$V_8(y,z) :- \texttt{C}(x,y,z) \qquad V_{12}() :- \texttt{C}(x,y,z)$$

Figure 4: All relational projections of `Contacts`

4. GENERATING LABELERS

The naïve disclosure labeling algorithm proposed in Section 3.3 runs in time that is linear in the size of the set $\mathcal{F}$. Unfortunately, $\mathcal{F}$ can easily become very large, since it generally contains all possible subsets of a set of security views $\mathcal{S}$. In fact, as the next example demonstrates, even $\mathcal{S}$ itself can grow quite large.

Example 4.1. Consider a generalization of the example used in Figure 3. Suppose we have an n-attribute relation R and wish to label each query over R with the set of relational projections on R that is required to answer it. There are 2^n possible projections on R - Figure 4 shows all of them for the three-attribute relation `Contacts` from Figure 1, where the relation name is abbreviated as `C`. The set $\mathcal{F}$ would need to account for all possible subsets of these projections, for a total size that is doubly-exponential in n.

It is clearly impractical to work with such a large $\mathcal{F}$. Fortunately, we do not need to represent all of $\mathcal{F}$ explicitly. We can instead work with a smaller subset of $\mathcal{F}$ and in some sense materialize any remaining elements as they are needed. This section focuses on the problem of finding a suitable subset of $\mathcal{F}$ which is as small as possible.

We assume the existence of two black-box algorithms that depend on $\preceq$ and on $\mathcal{U}$. The first takes as input subsets $\mathcal{W}_1$ and $\mathcal{W}_2$ of $\mathcal{U}$, and determines whether $\mathcal{W}_1 \preceq \mathcal{W}_2$ in the disclosure order. The second, written $GLB(\mathcal{W}_1, \mathcal{W}_2)$, finds a set of views $\mathcal{W}_3$ such that $(\Downarrow \mathcal{W}_1) \sqcap (\Downarrow \mathcal{W}_2) = (\Downarrow \mathcal{W}_3)$. GLB generalizes to handle an arbitrarily large number of input arguments in the obvious way. Section 5 contains a concrete instantiation of these algorithms for the case of equivalent view rewriting on conjunctive queries.

4.1 Downward generating sets

Suppose we have a set $\mathcal{F}$ that induces a labeler. This means it is closed under the GLB operation. Therefore, some elements of $\mathcal{F}$ are "redundant" in the sense that they can be computed by taking GLBs of other elements. Such redundant elements can be removed to yield a smaller $\mathcal{F}_d$ that can replace $\mathcal{F}$ for practical purposes.

Definition 4.2 (Downward generating set). *Given a set $\mathcal{F}$, we call $\mathcal{F}_d \subseteq \mathcal{F}$ a downward generating set for $\mathcal{F}$ if for every $\mathcal{W} \in \mathcal{F}$ there exist $\mathcal{W}_1, \mathcal{W}_2, \ldots, \mathcal{W}_n \in \mathcal{F}_d$ such that $\mathcal{W} \equiv GLB(\mathcal{W}_1, \mathcal{W}_2, \ldots, \mathcal{W}_n)$.*

It turns out every $\mathcal{F}$ that induces a labeler has a unique minimal downward generating set.

THEOREM 4.3. *If $\mathcal{F}$ induces a labeler then there exists a downward-generating set $\mathcal{F}_d$ for $\mathcal{F}$ which is minimal under the usual set ordering. The elements of $\mathcal{F}_d$ are uniquely determined up to equivalence.*

Given $\mathcal{F}$, a minimal downward generating set can be computed by iteratively removing elements of $\mathcal{F}$ that are equivalent to the GLB of a subset of the elements still left.

EXAMPLE 4.4. Continuing with Example 4.1, let $\preceq$ be the equivalent view rewriting ordering. A downward generating set for the original $\mathcal{F}$ is $\wp(\{V_3, V_6, V_7, V_8\})$. The reason for this will become clearer in Section 5 when we explain how GLB is computed for the equivalent view rewriting order. It turns out that

$$GLB(\{V_6\}, \{V_7\}) \equiv \{V_9\}$$
$$GLB(\{V_6\}, \{V_8\}) \equiv \{V_{10}\}$$
$$GLB(\{V_7\}, \{V_8\}) \equiv \{V_{11}\}$$
$$GLB(\{V_6\}, \{V_7\}, \{V_8\}) \equiv \{V_{12}\}$$

This should provide intuition as to why we removed the last four views in Figure 4. Note that the size of $\mathcal{F}_d$ is still exponential in the number of attributes of `Contacts`.

Given a downward generating set $\mathcal{F}_d$, the following procedure GLBLABEL shows how to use it for query labeling. The algorithm iterates over all elements of $\mathcal{F}_d$ (Line 3) and computes a running GLB of those elements that disclose at least as much information as $\mathcal{W}$ (Lines 4-6).

```
1: procedure GLBLabel(F_d, W)
2:     L ← ⊤
3:     for W' ∈ F_d do
4:         if W ⪯ W' then
5:             L ← GLB(L, W')
6:         end if
7:     end for
8:     return L
9: end procedure
```

Last but not least, downward generating sets obviate the problem of checking whether a given collection of sets of security views induces a labeler. It turns out that we can extend *any* set $\mathcal{G}$ to one that induces a labeler by closing it under the GLB operation.

THEOREM 4.5. *If $\mathcal{G}$ is a collection of sets of views that contains the top element ($\Downarrow \mathcal{U}$) then there is a set $\mathcal{F} \supseteq \mathcal{G}$ such that (i) $\mathcal{F}$ induces a disclosure labeler and (ii) $\mathcal{G}$ is a downward generating set for $\mathcal{F}$. The elements of $\mathcal{F}$ are unique up to equivalence.*

This is in some sense the converse of Theorem 4.3 and in practice removes the need for checking whether $\mathcal{G}$ induces a labeler. If it does not, we know we can extend it to one that does. In fact, we are free to work directly with $\mathcal{G}$ since it is a downward generating set for that labeler.

4.2 Generating sets

Although the reduction in size from $\mathcal{F}$ to $\mathcal{F}_d$ is substantial, Example 4.4 demonstrates that the size of $\mathcal{F}_d$ can still be exponential in the number of attributes in the database schema. The question arises whether we can find a smaller subset of $\mathcal{F}_d$ and use that for labeling instead. The answer is *yes*; however, we must place some restrictions on $\mathcal{F}$ and $\mathcal{U}$. We begin by defining and explaining these restrictions.

DEFINITION 4.6 (PRECISE LABELER). *Suppose $\mathcal{F}$ contains $\emptyset$, and additionally if $(\Downarrow \mathcal{W}_1) \in \mathcal{F}$ and $(\Downarrow \mathcal{W}_2) \in \mathcal{F}$ then $\Downarrow (\mathcal{W}_1 \cup \mathcal{W}_2) \in \mathcal{F}$. We say $\mathcal{F}$ induces a precise labeler.*

This definition states that $\mathcal{F}$ – or rather, its extension to the disclosure lattice on $\mathcal{U}$ – is closed under the lattice LUB operator. The intuition for the term *precise* is the following. Return to Figure 3 and suppose $\mathcal{F} = \{\emptyset, \{V_5\}, \{V_2\}, \{V_4\}, \top\}$, $\mathcal{U}$ is the set of all conjunctive queries over `Meetings` and $\preceq$ is the equivalent view rewriting order. $\mathcal{F}$ induces a labeler ℓ over $\mathcal{U}$, but it is not precise. Specifically, $\ell(\{V_2, V_4\}) = \top = \{V_1\}$, which is properly higher in the disclosure order than $\{V_2, V_4\}$, so the labeler exhibits some imprecision on this set of views.

The second concept we need relates to the universe $\mathcal{U}$ under the ordering $\preceq$.

DEFINITION 4.7 (DECOMPOSABILITY). *We say a set $\mathcal{U}$ is decomposable under $\preceq$ if for every $\mathcal{W}_1, \mathcal{W}_2 \subseteq \mathcal{U}$ and every $\{V\} \preceq \mathcal{W}_1 \cup \mathcal{W}_2$ we have either $\{V\} \preceq \mathcal{W}_1$ or $\{V\} \preceq \mathcal{W}_2$.*

The following Theorem is related to a result in [20].

THEOREM 4.8. *If $\mathcal{U}$ is decomposable under $\preceq$, then the corresponding disclosure lattice $\mathcal{I}$ is distributive.*

Assume now that $\mathcal{U}$ is decomposable under $\preceq$ and $\mathcal{F}$ induces a precise labeler. We can define the concept of a (full) *generating set* for $\mathcal{F}$.

DEFINITION 4.9 (GENERATING SET). *We say that $\mathcal{F}_{gen}$ is a **generating set** for $\mathcal{F}$ if every element of $\mathcal{F}$ is equivalent to the union of GLBs of elements of $\mathcal{F}_{gen}$.*

Analogues of Theorems 4.3 and 4.5 hold for generating sets. Given a set $\mathcal{F}$ that induces a precise labeler, a **minimal** generating set for $\mathcal{F}$ always exists, and is guaranteed to be unique up to equivalence. Conversely, we can extend any set $\mathcal{G}$ to an $\mathcal{F}$ that induces a precise labeler and for which $\mathcal{G}$ is a generating set. $\mathcal{F}_{gen}$ is generally much smaller than either $\mathcal{F}$ or $\mathcal{F}_d$, although of course it only exists under the two restrictions we outlined above.

EXAMPLE 4.10. Continuing with Example 4.4, suppose $\mathcal{U}$ is decomposable under the equivalent view ordering $\preceq$; this is true for instance if we take $\mathcal{U}$ to be the set of all *single-atom* queries over `Contacts`. In this case, the set $\mathcal{F}_{gen} = \{\{V_3\}, \{V_6\}, \{V_7\}, \{V_8\}\}$ is a generating set for a $\mathcal{F}$ that induces a precise labeler over $\mathcal{U}$. The size of $\mathcal{F}_{gen}$ is now only linear in the number of attributes of `Contacts`.

Given a generating set $\mathcal{F}_{gen}$, we can use it for query labeling. The following algorithm processes $\mathcal{W}$ one view at a time (Line 3) and computes a running union of the labels for the views (Line 4).

```
1: procedure LabelGen(F_gen, W)
2:     result ← ∅
3:     for each V ∈ W do
4:         result ← result ∪ GLBLabel(F_gen, {V})
5:     end for
6:     return result
7: end procedure
```

The takeaway is that if $\mathcal{U}$ is decomposable and we desire a precise labeler, it is easy to label queries using a set of security views $\mathcal{S}$. We can simply use the set $\{\{S_i\} \mid S_i \in \mathcal{S}\}$, consisting of singleton sets containing each view in $\mathcal{S}$, as our $\mathcal{F}_{gen}$ and run algorithm LABELGEN to perform a labeling.

5. LABELING CONJUNCTIVE QUERIES

We now use the theory and results from Sections 3 and 4 and show how to label a particular class of queries. We focus on labeling conjunctive queries with set semantics under the equivalent view rewriting order, using a set of single-atom security views. Although not all of the security views that would be useful for real-world systems can be modeled without joins, a large fraction can. Extending these algorithms to multi-atom security views is ongoing work.

We write $\mathcal{U}_{atom}$ to denote the set of single-atom conjunctive views defined over a given database schema, and $\mathcal{U}_{cv}$ to denote the set of all conjunctive views. Let $\preceq$ be the equivalent view rewriting order, and assume we have a set of single-atom security views $\mathcal{S}$. We explain how to label arbitrary conjunctive queries with subsets of $\mathcal{S}$.

We find it useful to work with a modified representation of conjunctive queries where we associate each query with a list of its body atoms and discard the head. To keep track of which of the variables are distinguished and which are existential, we tag them accordingly. For example, the query Q_2 from Figure 1 is represented as $[\mathtt{M}(x_d, y_e), \mathtt{C}(y_e, w_e, \mathtt{'Intern'})]$, where the subscripts e and d denote existential and distinguished variables respectively and `M` and `C` abbreviate `Meetings` and `Contacts` respectively.

We present the process of labeling in two stages. First, we explain how sets of single-atom queries may be labeled, and then extend the process to sets of multi-atom queries.

5.1 Single-atom case

$\mathcal{U}_{atom}$ is decomposable. Consequently, the discussion and labeling algorithm from Section 4.2 apply directly. The set $\{\{S_i\} \mid S_i \in \mathcal{S}\}$, composed of singleton sets containing each of the security views, serves as a generating set for the labeler. For a complete end-to-end labeling algorithm, we only need to define implementations of the two subroutines introduced at the beginning of Section 4. The first determines, given $\mathcal{W}_1, \mathcal{W}_2 \subseteq \mathcal{U}_{atom}$, whether $\mathcal{W}_1 \preceq \mathcal{W}_2$. The second computes the GLB function – that is, given $\mathcal{W}_1, \mathcal{W}_2$ it finds a $\mathcal{W}_3$ such that $(\Downarrow \mathcal{W}_1) \sqcap (\Downarrow \mathcal{W}_2) = (\Downarrow \mathcal{W}_3)$.

Determining whether $\mathcal{W}_1 \preceq \mathcal{W}_2$ can be done using standard techniques from the literature on equivalent view rewriting [10]. It remains to show how to compute the GLB.

The key to computing GLB is a procedure GLBSingleton for computing the GLB of two singleton sets of views $\{V_1\}$ and $\{V_2\}$; this can be extended to multi-element sets of views in a manner to be explained shortly.

GLBSingleton is based on the idea of unification. It begins by computing a generalized *most general unifier* (mgu) [6] of the the bodies of V_1 and V_2. This is computed by a subroutine called GenMGU, which differs from a standard mgu computation in three ways. First, if the algorithm attempts to unify a constant with an existential variable, the unification fails. Second, if the algorithm attempts to unify an existential variable with an existential or distinguished variable, the result is an existential variable. Third, if the algorithm attempts to unify two distinguished variables, the result is another distinguished variable. We explain these differences using some examples.

First, we show why the unification of a constant with an existential variable must fail.

Example 5.1. Consider the following boolean views:

$$V_{13}() :- \mathtt{M}(\mathtt{9}, \mathtt{'Jim'}) \qquad V_{14}() :- \mathtt{M}(x, y)$$

The first view tests whether `Meetings` contains a particular tuple and the second checks whether it contains any tuples at all. The standard mgu of the body atoms is equal to the first atom, but the actual GLB of the views should be $\bot$. There is no single-atom query that can be rewritten in terms of V_{13} and also in terms of V_{14}.

Next, we illustrate the reasons for our handling of existential and distinguished variables.

Example 5.2. Consider views V_6 and V_7 from Figure 4:

$$V_6(x, y) :- \mathtt{C}(x, y, z) \qquad V_7(x, z) :- \mathtt{C}(x, y, z)$$

In our new representation they become $[\mathtt{C}(x_d, y_d, z_e)]$ and $[\mathtt{C}(x_d, y_e, z_d)]$ respectively. Their GenMGU is $[\mathtt{C}(x_d, y_e, z_e)]$, i.e. V_9 from Figure 4. This makes intuitive sense as V_9, the projection on the first attribute of `Contacts`, accurately represents the overlap between V_6 and V_7, i.e. the information that can be computed from either V_6 or V_7 in isolation.

Once GenMGU is available, an extra check is needed to rule out some corner cases as shown in the next example.

Example 5.3. Consider the following boolean views:

$$V_{14}() :- \mathtt{M}(x, y) \qquad V_{15}() :- \mathtt{M}(z, z)$$

The GenMGU of the body atoms is $[\mathtt{M}(w_e, w_e)]$, but the GLB should be $\bot$ by the same reasoning as in example 5.1.

The check to eliminate such cases is conceptually straightforward. It involves finding situations where computing GenMGU forces a new equality constraint on two values in the same original atom, and where at least one of these values was an existential variable. If we find such a situation or if GenMGU fails, GLBSingleton returns $\bot$; otherwise it returns the output of GenMGU.

GLBSingleton can be extended to non-singleton sets for a complete implementation of GLB($\mathcal{W}_1, \mathcal{W}_2$). We simply compute the pairwise GLBSingleton of singleton sets containing each pair of views $V_1 \in \mathcal{W}_1, V_2 \in \mathcal{W}_2$ and union all the results together. This completes the description of GLB, giving us the last tool we need to label queries using the techniques from Section 4.2.

5.2 Multi-atom case

The set $\mathcal{U}_{cv}$ of arbitrary conjunctive queries is not in general decomposable; therefore, we are unable to use the same techniques as above. However, because we have restricted the set $\mathcal{S}$ to contain single-atom views only, we can perform labeling efficiently by solving the problem in two steps. To label a set of queries $\mathcal{Q} \subseteq \mathcal{U}_{cv}$, we first convert each $Q \in \mathcal{Q}$ into a set of single-atom queries using the Dissect algorithm described below. In the second step, we compute the disclosure label of the resulting set of single-atom views using the algorithm discussed in the previous subsection.

The Dissect algorithm begins by computing a *folding* [9] of Q, which intuitively removes "redundant" atoms from Q. Next, it splits up the folding of Q into its constituent atoms, except that any existential variable that appears in at least two atoms is promoted to a distinguished variable.

Example 5.4. Consider query Q_2 from Figure 1, i.e. $[\mathtt{M}(x_d, y_e), \mathtt{C}(y_e, w_e, \mathtt{'Intern'})]$. The result of running Dissect on this query is a set that contains two single-atom queries: $[\mathtt{M}(x_d, y_d), \mathtt{C}(y_d, w_e, \mathtt{'Intern'})]$.

Intuitively, the reason we need to promote existential variables to distinguished ones is that we are labeling with single-atom views. Recall that the set of single-atom views in a query's disclosure label must contain enough information to uniquely determine the query's answer. Any set of single atom security views that allows a join to be computed must reveal the values of the join attributes.

We can show that DISSECT is a disclosure labeler with domain $\wp(\mathcal{U}_{cv})$ and image $\wp(\mathcal{U}_{atom})$. As the composition of two labelers is also a labeler, we can create a disclosure labeler for multi-atom conjunctive queries by combining DISSECT with our single-atom labeling procedure.

6. IMPLEMENTATION

At this point, we have introduced our new notion of a disclosure labeler that is data-derived, semantically meaningful and expressive, and we have presented practical algorithms for labeling conjunctive queries. In this section, we describe two key optimizations that allow us to efficiently manage complex security policies for regulating the cumulative disclosure of information over time. First, we store disclosure labels in a heavily compressed format that makes comparisons between different disclosure labels very fast. And second, we represent security policies in a way that allows us to make policy decisions without ever needing to refer back to a list of previously executed queries.

6.1 Representing disclosure labels

We begin by revisiting the GLBLABEL disclosure labeling algorithm from Section 4.1. In its simplest form GLBLABEL takes as input a set of security views $\mathcal{F}_{gen}$ and a singleton set $\{V\}$ whose disclosure label we wish to find, and returns the GLB of the following collection of singleton sets:

$$\{\{V_i\} : V_i \in \mathcal{F}_{gen} \text{ and } \{V\} \preceq \{V_i\}\}$$

In practice, however, computing the GLB is completely unnecessary. Instead, we compute

$$\ell^+(\{V\}) = \{V_i \in \mathcal{F}_{gen} : \{V\} \preceq \{V_i\}\}$$

Roughly speaking, this is the set of all security views that uniquely determine the answer to V. If we know $\ell^+(\{V\})$, we can compute $\ell(\{V\})$. Furthermore, we can now efficiently compare the disclosure labels of two different points $\ell(\{V\}$ and $\ell(\{V'\})$ in the lattice of disclosure labels:

$$\ell(\{V\}) \preceq \ell(\{V'\}) \text{ if and only if } \ell^+(\{V\}) \supseteq \ell^+(\{V'\})$$

We provide an example in order to solidify this idea:

EXAMPLE 6.1. Continuing Example 4.10, let

$$\mathcal{F}_{gen} = \{\{V_3\}, \{V_6\}, \{V_7\}, \{V_8\}\}$$

The disclosure label for $\{V_9\}$ is $GLB(\{V_3\}, \{V_6\}, \{V_7\})$, and consequently, $\ell^+(\{V_9\}) = \{V_3, V_6, V_7\}$. Similarly, the disclosure label for $\{V_{12}\}$ is $GLB(\{V_3\}, \{V_6\}, \{V_7\}, \{V_8\})$, so that $\ell^+(\{V_{12}\}) = \{V_3, V_6, V_7, V_8\}$. Examining these two sets, it is clear that $\ell^+(\{V_{12}\}) \supseteq \ell^+(\{V_9\})$, and we therefore conclude that $\ell(V_{12}) \preceq \ell(V_9)$.

In practice, these subsets of $\mathcal{F}_{gen}$ can be represented as bit vectors; we use bit mask operations to determine whether one subset contains another. Since $\{V_1\} \preceq \{V_2\}$ only if V_1 and V_2 are views over the same base relation, we can further optimize this representation. In our current implementation, the low 32 bits of a 64-bit integer track which base relation a view corresponds to, and the remaining 32 bits represent the elements of $\mathcal{F}_{gen}$ that are associated with that relation. In this way, a single 64-bit integer can store a disclosure label for a disclosure lattice with up to 2^{32} distinct relations, each of which is associated with 32 distinct elements from $\mathcal{F}_{gen}$. There is nothing special about the number 32, and the representation can easily be generalized to any number of bits.

We extend this representation to multi-atom disclosure labels by using arrays of single-atom disclosure labels. For instance, in Example 6.1, the disclosure label of $\{V_6, V_7\}$ can be stored as a two-element array whose first element is $\ell^+(\{V_6\})$ and whose second element is $\ell^+(\{V_7\})$.

Complexity Analysis: Let n denote the number of atoms in the input query, and m denote the number of security views. The DISSECT algorithm from Section 5.2 relies on query folding as a subroutine. Query folding is known to be NP-hard, and our current implementation uses a brute-force search that runs in time that is exponential in n. Dissection yields a set of at most n single-atom views, and in the worst case, the labeler must determine whether each of the n views returned by DISSECT can be rewritten in terms of each of the m security views, for a total of $O(n \cdot m)$ comparisons. Each comparison can be performed in time linear in the total size of its two input atoms. Once they have been computed, the disclosure labels of an r-atom query and an s-atom query can be compared in time $O(r \cdot s)$.

6.2 Representing security policies

Formally, a security policy is defined in Section 3.4 as a subset of the elements in a labeler's disclosure lattice. However, as we have already noted, disclosure lattices can become enormous even for small databases; storing security policies explicitly is therefore impractical. In this section, we discuss a different representation of security policies that drastically reduces space consumption. As an added bonus, we are able to track and restrict cumulative disclosure with very little space or computational overhead. We restrict our discussion to a system with a single principal; a generalization to multiple principals is straightforward.

Let us first consider the simpler problem of enforcing a stateless security policy. When a principal issues a query Q, a reference monitor decides whether to answer or refuse the query based solely on the query's disclosure label and on the security policy itself. In this model, a security policy can be represented as a set $\mathcal{W}$ of disclosure labels for single-atom views; a query is answered if its disclosure label is below $\mathcal{W}$, and is refused otherwise.

We now discuss a variant of this algorithm that limits cumulative information disclosure over time. When a principal issues a query Q_n, a reference monitor looks at both Q_n and the list of previously answered queries $Q_1, Q_2, \ldots, Q_{n-1}$. In the case where $\{Q_1, Q_2, \ldots, Q_n\} \preceq \mathcal{W}$, the query is answered. Otherwise, the query is refused.

Crucially but perhaps counterintuitively, the two models described above are actually equivalent. Formally, the first model ensures that $\{Q_i\} \preceq \mathcal{W}$ for each $i = 1, 2, \ldots, n$. The second model ensures that $\{Q_1, Q_2, \ldots, Q_n\} \preceq \mathcal{W}$. Equivalence follows immediately from the definition of a disclosure order (Definition 3.1).

What this means in practice is that even a stateless reference monitor can restrict cumulative information disclo-

sure. Unfortunately, this guarantee comes at a cost: it is no longer possible to represent *stateful* security policies, such as the *Chinese Wall* policies required by many business applications, with this model. In order to support such policies, we represent a security policy as a *collection* of sets of single-atom disclosure labels, say $\{\mathcal{W}_1, \mathcal{W}_2, \ldots, \mathcal{W}_k\}$. We enforce the invariant that if $Q_1, Q_2, \ldots, Q_n$ are the queries that have been answered so far then $\{Q_1, Q_2, \ldots, Q_n\} \preceq \mathcal{W}_i$ must hold for *some* $\mathcal{W}_i$. We refer to each $\mathcal{W}_i$ as a *partition* of the security policy.

EXAMPLE 6.2. Consider the security policy $\{\mathcal{W}_1, \mathcal{W}_2\}$ in which $\mathcal{W}_1 = \{V_1\}$ and $\mathcal{W}_2 = \{V_3\}$. This policy encodes the constraint that a principal may access either `Meetings` or `Contacts`, but not both. If the principal Alice issues query V_6, this query will be accepted, since $\{V_6\} \preceq \mathcal{W}_2$. If she next issues the query V_7, this query will also be accepted, since $\{V_6, V_7\} \preceq \mathcal{W}_2$. However, if she then issues the query V_2, this query will be refused, since $\{V_6, V_7, V_2\} \not\preceq \mathcal{W}_1$ and $\{V_6, V_7, V_2\} \not\preceq \mathcal{W}_2$.

A naïve implementation of this scheme would require us to search through a principal's entire query history whenever we make a policy decision for a new query. Fortunately, this is not necessary. In fact, we only need to keep track of which of the $\mathcal{W}_i$ are consistent with all the queries answered so far; we can do so with a bit vector that contains one bit for each partition of the policy.

EXAMPLE 6.3. In Example 6.2, the reference monitor's bit vector is initially $\langle 1, 1\rangle$, which indicates that $\emptyset \preceq \mathcal{W}_1$ and $\emptyset \preceq \mathcal{W}_2$. After answering V_6, the bit vector becomes $\langle 1, 0\rangle$ because the $\{V_6\} \preceq \mathcal{W}_1$ but $\{V_6\} \not\preceq \mathcal{W}_2$. The bit vector is left unchanged after the second query. If the third query was answered, the bit vector would become $\langle 0, 0\rangle$ to indicate that $\{V_6, V_7, V_2\}$ is not below either $\mathcal{W}_1$ or $\mathcal{W}_2$ in the lattice of disclosure labels. However, the reference monitor will instead refuse the query and leave the bit vector as $\langle 1, 0\rangle$.

7. LABELING IN PRACTICE

In this Section, we showcase the practical applicability and usefulness of our labeling-based disclosure control techniques. We begin by presenting results of a manual review of an existing labeling-based disclosure control system – the permissions structure associated with Facebook's Graph API and FQL. Next, we present experimental results from an implementation of our labeling algorithms. We conclude by evaluating the throughput of the policy checker described in Section 6.2.

7.1 Reviewing Facebook's APIs

Facebook provides two APIs through which apps can query user data: the Graph API and FQL. They also define a set of permissions such as `user_likes` and `friends_likes`, each of which grants an app access to a particular view over the data. Before an app can issue an API query, it must request access to a specific set of permissions. In our terminology, a set of permissions corresponds to a *disclosure label*, and each Facebook app is effectively a different principal.

Facebook's developer documentation specifies the minimal set of permissions needed to execute different API queries. In other words, it provides a hand-generated disclosure label for each of these queries. We hypothesized that as the APIs grow larger and more complex, manual labeling becomes error-prone. We identified 42 different views over the `User` table accessible through both APIs and compared the respective permissions as given in the documentation. That is, we identified pairs of corresponding queries in both APIs where both queries selected a particular attribute of the `User` table. We then compared the required permissions listed in the documentation for each pair of queries.

We found discrepancies in the permissions needed for six of the 42 views; details are shown in Figure 2. This illustrates the difficulty of manually labeling queries. Our findings are consistent with previous studies of human-generated query labels in a different setting, namely Android apps [16].

For each of the six Facebook views mentioned above, we issued appropriate queries in both APIs to determine which permissions were really required. In all six cases, we found that the same query in both APIs required the same permissions, as shown for each query in the last column of Figure 2. Thus, the inconsistencies were in the documentation only. Nonetheless, such errors are alarming. Tracking complex permission structures by hand is challenging, and the chances that developers will select the wrong permissions for their apps are compounded if they rely on inaccurate documentation.

7.2 Experimental evaluation

We implemented and evaluated two key systems: a disclosure labeler for multi-atom queries, and a mechanism that makes policy decisions based on the labeler's output. The disclosure labeler, which emphasized scalability but not raw performance, was implemented in Java, and was tested with the Java 1.7 VM. The policy mechanism was implemented in C and compiled with GCC 4.2. All our tests were conducted on a laptop with a 2.9GHz Intel Core i7 processor running Mac OS X 10.8. Our benchmarks measured process rather than wall time.

Our test database contained eight different relations that captured core functionality from the Facebook API. The largest of these was the `User` relation, which contained 34 distinct attributes. Each of the remaining relations contained between 3 and 10 attributes.[3]

For each relation, we selected a set of security views that could support the confidentiality policies described in Facebook's developer documentation. The most complex relation, the `User` relation, required us to define a generating set $\mathcal{F}_{gen}$ with 16 distinct security views; most of the other relations we considered could be modeled using just three views. The main difficulty that we encountered was that some of the permissions that Facebook uses require a notion of joins. For instance, there is a permission that allows a Facebook app to see the birthdays of all of a user's Facebook friends. Formally, this can be modeled using a join between the `User` relation and the `Friend` relation. Our current implementation does not yet support security views that have joins in them. We dealt with this issue by adding an extra column to each relation that indicated whether the owner of a given tuple was friends with the principal executing the query. Since the list of a user's friends is available to any

[3]In preliminary tests on synthetic data, we tried increasing the total number of relations to 1,000 while keeping the number of security views per relation constant; the total number of relations did not have any appreciable impact on the hash-based disclosure labelers' throughput.

Attribute	FQL Permissions	Graph API Permissions	Correct Labeling
pic ("picture" in Graph API)	none	any for pages with whitelisting/targeting restrictions, otherwise none	FQL
timezone	any	Available only for the current user	Graph API
devices	any	any; only available for friends of the current user	Graph API
relationship_status	any	`user_relationships` or `friends_relationships`	Graph API
quotes	`user_likes` or `friends_likes`	`user_about_me` or `friends_about_me`	FQL
profile_url ("link" in Graph API)	any	none	FQL

Table 2: Inconsistencies between the FQL and Graph API permissions labeling of `User` attributes; "any" means any nonempty set of permissions and "none" means no permissions are required

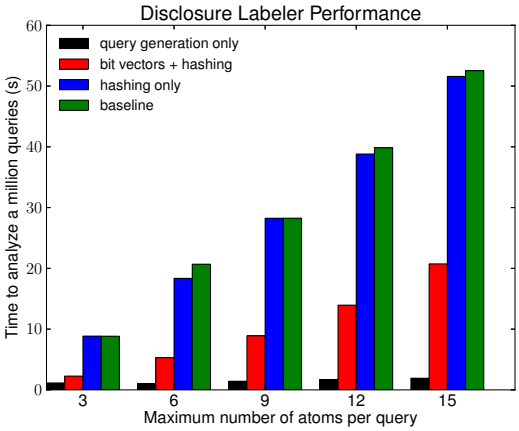

Figure 5: Disclosure labeler performance.

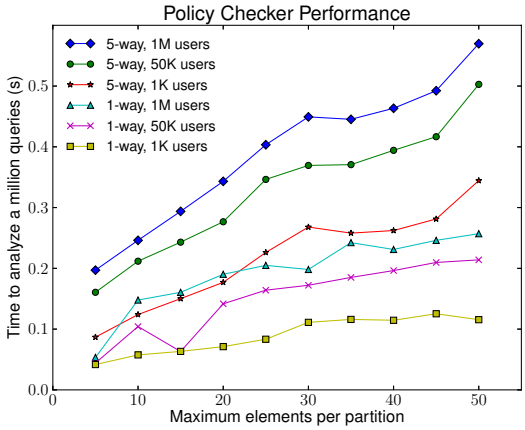

Figure 6: Policy checker performance. The top-to-bottom ordering in the legend mirrors the top-to-bottom ordering of the curves.

app running on behalf of that user, this denormalization did not affect the accuracy of our model.

After examining a number of sample Facebook applications, we decided to use a workload of queries that were randomly generated with the following process:

1. Select a random relation from the schema.
2. Select a random subset of its attributes.
3. Randomly request these attributes for either (i) the current user, (ii) friends of the current user, (iii) friends of friends of the current user, or (iv) a non-friend.

In Step (3) above, we note that Option (ii) involved a join with the `Friend` relation, and Option (iii) involved *two* joins with the `Friend` relation. Hence, each query contained between one and three body atoms. In order to stress-test our algorithm, we extended our workload to generate (unrealistically) complex queries; we did this by repeating the process above between one and five times, and joining the resulting subqueries on the `uid` (User ID) attribute, which appeared in all the relations we considered.

We used this workload to evaluate the performance of three different versions of our disclosure labeling algorithm. The first version, which we used as a baseline, was a straightforward adaptation of the LabelGen algorithm from Section 4.2. The second version used a hashtable to partition views based on the relation they referenced. The third version made use of both hashtable partitioning and the bit vector optimization from Section 6.1. In a fourth experiment, we considered only the time needed to randomly generate parsed queries but not to label them.

Results are shown in Figure 5. The labeler that only incorporated hashing generally outperformed the baseline by a small margin. The labeler that additionally made use of the bit vector optimization consistently outperformed both of them by a factor of 3x to 4x – it was able to process a million queries, each with between 1 and 3 body atoms, in slightly more than 2 seconds.

We hypothesized that the optimizations from Section 6 would make it possible to reason about complex security policies very efficiently once the disclosure label for a query was known. To verify this hypothesis, we wrote a simple policy checker that maintained information about the security policies of between 1,000 and 1,000,000 distinct principals. Each principal's security policy was randomly generated. The maximum number of partitions per policy was set to either 1 (a stateless security policy) or 5 (a fairly complex Chinese Wall policy). However, the actual number of partitions per policy could vary between principals, reflecting the intuition that some principals would require more complex policies than others. Similarly, we allowed the maximum number of elements (i.e., single-atom views) per partition to vary between 5 and 50. Intuitively, we should expect the number of security views to increase as users define fine-grained policies to control how their data is shared.

We ran our experiment on a collection of 10 million disclosure labels output by the previous experiment. Each labeled query contained between one and three body atoms. Queries were randomly assigned to principals, and the appropriate policy was enforced for each principal.

The results are shown in Figure 6. For relatively simple policies, the analysis time for a million queries was between 0.04 and 0.15 seconds, depending on the number of principals in the system. Throughput decreased very gradually as the number of principals in the system increased. This decrease was likely due to issues with cache locality: as the number of principals grew larger, it became increasingly improbable that the metadata for a randomly selected principal would reside in an on-chip cache. For a million principals, our system was able to analyze a million disclosure labels in about 0.57 seconds on the most complex policies we tested.

8. RELATED WORK

We have already discussed some related work, particularly on the topic of enforcing security policies through equivalent view rewriting. An alternative approach is to semantically modify the original query into a new query that is safe to be executed, as in Oracle's *Virtual Private Database* and similar systems [24]. In such systems, the query that is answered may not be the same as the one that a principal issues [27]. Miklau and Suciu propose a probabilistic notion of *non-disclosure* in database systems [22]. Their work is in some sense dual to ours: whereas they are able to enforce strong guarantees about the information that a query does *not* disclose, our approach is able to capture nuances of the information that a query *does* disclose. The authors of [19] develop a number of axioms for *query pricing* that have parallels to our requirements for disclosure labelers. However, because the authors explicitly decouple the relative pricing of queries from the relative amount of information disclosed by these queries, our approach is not directly comparable with theirs. There are also connections between our labeling approach and data integration work on *universal solutions* [15]. Exploring these connections in depth is future work.

Acknowledgments. This research was supported by Grant 279804 of the European Research Council, the National Science Foundation under Grants IIS-1012593 and IIS-0911036, by the iAd Project funded by the Research Council of Norway, and by a Google Research Award and an NEC Research Award. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

9. REFERENCES

[1] Box OneCloud. `http://developers.box.com/onecloud/`.

[2] Facebook Developers. `https://developers.facebook.com`.

[3] Overwarch SoldierEyes. `http://www.overwatch.com/products/soldiereyes.php`.

[4] Personal data ecosystem consortium. `http://pde.cc`.

[5] What they know – Mobile. `http://blogs.wsj.com/wtk-mobile/`.

[6] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[7] F. N. Afrati, C. Li, and J. D. Ullman. Generating efficient plans for queries using views. *SIGMOD Rec.*, 30, May 2001.

[8] D. Brewer and M. Nash. The chinese wall security policy. In *IEEE Symposium on Security and Privacy*, 1989.

[9] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, 1977.

[10] M. Compton. Finding equivalent rewritings with exact views. In *ICDE*, 2009.

[11] B. Davey and H. Priestley. *Introduction to lattices and order*. Cambridge university press, 2002.

[12] C. Dwork. Differential privacy. In *Automata, Languages and Programming*, volume 4052 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006.

[13] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in iOS applications. In *NDSS*, 2011.

[14] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[15] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1), 2005.

[16] A. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In *USENIX 2011*, 2011.

[17] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: user attention, comprehension, and behavior. In *SOUPS*, 2012.

[18] G. Gou, M. Kormilitsin, and R. Chirkova. Query evaluation using overlapping views: completeness and efficiency. In *SIGMOD*, 2006.

[19] P. Koutris, P. Upadhyaya, M. Balazinska, B. Howe, and D. Suciu. Query-based data pricing. In *PODS*, 2012.

[20] J. Landauer and T. Redmond. A lattice of information. In *CSFW*, 1993.

[21] A. Y. Levy, A. O. Mendelzon, and Y. Sagiv. Answering queries using views (extended abstract). In *PODS*. ACM, 1995.

[22] G. Miklau and D. Suciu. A formal analysis of information disclosure in data exchange. *J. Comput. Syst. Sci.*, 73(3), 2007.

[23] A. Nash, L. Segoufin, and V. Vianu. Views and queries: Determinacy and rewriting. *ACM Trans. Database Syst.*, 35, July 2010.

[24] L. E. Olson, C. A. Gunter, and P. Madhusudan. A formal framework for reflective database access control policies. In *ACM CCS*, 2008.

[25] N. Perlroth. Linkedin's leaky mobile app has access to your meeting notes. New York Times, June 5 2012.

[26] R. Pottinger and A. Halevy. Minicon: A scalable algorithm for answering queries using views. *The VLDB Journal*, 10, 2001.

[27] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD*, 2004.

[28] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[29] D. A. Schultz. *Decentralized Information Flow Control for Databases*. Dissertation, MIT, 2012.