Explainable Security for Relational Databases

Gabriel Bender[*]
Cornell University
Ithaca, NY 14853, USA
gbender@cs.cornell.edu

Łucja Kot
Cornell University
Ithaca, NY 14853, USA
lucja@cs.cornell.edu

Johannes Gehrke
Cornell University
Ithaca, NY 14853, USA
johannes@cs.cornell.edu

ABSTRACT

Companies and organizations collect and use vast troves of sensitive user data whose release must be carefully controlled. In practice, the access policies that govern this data are often fine-grained, complex, poorly documented, and difficult to reason about. As a result, principals frequently request and are granted access to data they never use.

To encourage developers and administrators to use security mechanisms more effectively, we propose a novel security model in which all security decisions are formally *explainable*. Whether a query is accepted or denied, the system returns a concise yet formal explanation which can allow the issuer to reformulate a rejected query or adjust his/her security credentials. Our approach has a strong formal foundation based on previously unexplored connections between *disclosure lattices* and *policy algebras*. We build on this foundation and implement a disclosure control system that handles a wide variety of real SQL queries and can accommodate complex policy constraints.

> *"Smelling isn't everything" said the Elephant.*
> *"Why," said the Bulldog, "if a fellow can't trust his nose, what is he to trust?"*
> *"Well, his brains perhaps," she replied mildly.*
>
> – C.S. Lewis, *The Magician's Nephew*

Categories and Subject Descriptors

H.2 [**Information Systems**]: Database Management

Keywords

database security; view rewriting; access control

1. INTRODUCTION

Over the past few decades we have seen the emergence of companies like Google and Facebook that collect and store large amounts of personal user information. This data can be used to deliver highly relevant content to end users. It can also be used to match advertisements with users' interests, thereby creating significant value for advertisers.

However, companies risk serious damage to their reputations if they lose control over this data. Fine-grained permissions provide better control over who can access what resources, and as a result, platforms such as Google Android and Facebook Apps have dozens of distinct permissions, each regulating access to a different resource or type of user data. Similarly, commercial solutions such as IBM LBAC [5] and Oracle VPD [6] give database administrators precise control over which principals can see which parts of a database. Such control is largely motivated by the Principle of Least Privilege [23], which states that principals should be granted the least permissions needed to do their jobs.

Unfortunately, this flexibility comes with a high cost: fine-grained permissions structures are inherently complex and difficult to reason about. To make matters worse, the documentation for these permissions structures is often inaccurate or buggy. [8, 16] If a query fails due to insufficient permissions, the path of least resistance is to blindly request more permissions until the problem goes away. As a result, principals frequently violate the Principle of Least Privilege by acquiring permissions that they never use. [16]

In order to address this problem, we propose a new security model that is designed to ensure that all policy decisions are *explainable*. Instead of simply rejecting unauthorized queries, our system provides the issuer with concise *explanation* of why the queries were rejected and what additional permissions would need to be granted for a successful execution. The principal can then refine the queries or request additional permissions based on the explanation. In addition to its ability to generate explanations, our approach provides *strong formal security guarantees*, yet is able to *handle real SQL queries and security policies.*

Before presenting our work in more depth, we introduce our system model, overview existing disclosure control approaches, and justify why a new solution is needed.

1.1 Enforcing Security Policies

We begin with a generic architecture for access control that is applicable to many modern DBMSs. We clarify how our *explainable* model compares to traditional database security models and identify concrete properties that should be satisfied by explanations for policy decisions.

The example database shown in Figure 1 is based on the schema that Facebook exposes to third party app developers through the Facebook Query Language (FQL) [3]. The

[*]Work done while at the University of Washington

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.
Copyright 2014 ACM 978-1-4503-2376-5/14/06 ...$15.00.

User

UID	Name	Hobby
1	Babbage, Charles	math
2	Church, Alonzo	math
3	Lovelace, Ada	music
4	Turing, Alan	chess
5	von Neumann, John	history

Friend

UID1	UID2
1	3
3	1
2	4
4	2
4	5
5	4

```
CREATE VIEW V1 AS
  (SELECT name, hobby FROM User WHERE uid = 1);

CREATE VIEW V2 AS
  (SELECT name FROM User WHERE uid = 1);

CREATE VIEW V3 AS
  (SELECT U.uid, U.name FROM User U, Friend F
    WHERE U.uid = F.uid2 AND F.uid1 = 1);
```

Figure 1: Database and security views.

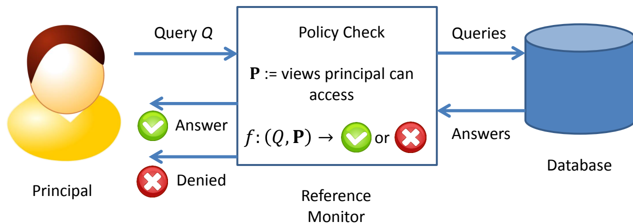

Figure 2: Disclosure control system model

schema contains two tables: `User`, which has a User ID (UID), Name, and Hobby for each user in the dataset, and `Friend`, which tracks friendship relations between users.

The system controls disclosure of this data using a *reference monitor* (RM) that enforces a *security policy.* At the time that the database schema is designed, a trusted system administrator defines a set of *security views* that reveal known types of information about the dataset. All subsequent policy decisions are made with the help of these security views. In Figure 1, `V1` reveals the UID, Name, and Hobby of User 1 (Charles Babbage) while `V2` reveals his UID and Name but not his Hobby and `V3` reveals the UIDs and Names of his friends. In a real system, an analogous set of security views would be generated for each end user; appropriate security views can easily be generated from templates defined by a human administrator. Many permissions used by Facebook [4] and Android [2] are essentially simple security views, though they are not formally defined in SQL.

The RM sits between the query evaluation engine and the principals who issue queries (Figure 2). When a principal issues a database query Q, the RM performs a policy check in order to determine whether the query should be accepted or rejected. This policy check is divided into two steps.

In the first step, the RM computes the subset $\mathbf{P}$ of the security views to which the principal has been granted access. An untrusted third-party app running on Charles Babbage's behalf might be permitted to see the contents of `V2` but not of `V1` or `V3`. In practice, this computation might be based on an *Access Control List* (ACL), a set of *Credentials* supplied by the issuer, or a *Role-Based* security mechanism.

In the second step, the RM checks whether the views in $\mathbf{P}$ contain enough information to answer the principal's original query. The determination is made by a *policy function* f that takes Q and $\mathbf{P}$ as inputs and returns 0 or 1. If f returns 1 then the principal receives the query results; otherwise, the query is denied. The function f is said to be *database-dependent* if its output can depend on the current state of the database, or *database-independent* otherwise. Our focus in this work is on *database-independent* policies; this ensures that neither policy decisions nor the corresponding explanations can leak potentially sensitive information about the current state of the database.

Ideally, f would be based on a simple mathematical criterion: it would determine whether the security views that the principal had access to contained enough information to answer Q. If so, f would output 1. Otherwise, it would output 0. In practice, however, f is typically a very crude and conservative algorithm.

In traditional implementations of view-based security, the onus is on the principal to write the query in terms of the views he or she can access. Let us return to Figure 1; suppose an app that has been granted access to `V2` but not `V1` issues the following query:

```
SELECT name FROM V1;
```

This query will be rejected even though it only reveals information that the app is allowed to access – specifically, it reveals the name of User 1, which can be computed from `V2`. The function f is not *data-derived* [8]: it checks whether the principal is allowed to access all the views in the `FROM` clause, but it does not account for the information that the query and the views reveal about the underlying database.

When the query is rejected it would be helpful for the RM to point out that the query can be rewritten using `V2` – which the principal has access to – in the following manner:

```
SELECT name FROM V2;
```

Better yet, the reference monitor should be able to search for rewritings automatically, and use them – if found – whenever it enforces a policy or generates an explanation.

It is not that we are lacking a formal theory of disclosure or the ability to compute rewritings; in previous work [8] we introduced a formal framework for reasoning about information disclosure in view-based security. However, this research is theoretical and does not deal with expressive real-world query languages and policies. In practice, explanations for rejected queries are often insufficient or nonexistent [20]. Even worse, systems such as Oracle's Virtual Private Database [6] accept unauthorized queries and modify their execution semantics without users' knowledge [22].

In summary, there is a fundamental connection between using a *data-derived* policy function f and the ability to provide explanations. Theoretical work shows how to compute data-derived functions f but does not extend to real-world systems; policy functions used in practice are not data-derived, which leads to poor explainability.

1.2 Our Contributions

In this paper, we introduce a novel instance of view-based security in which every policy decision made by the RM comes with a *concise*, *data-derived*, and *mathematically rigorous explanation.* Our work bridges the gap between existing theoretical work on disclosure lattices [8] and the needs of practical systems which must deal with complex SQL queries and sophisticated security policies.

In our system, both policy decisions and explanations are based on Boolean formulas in a *policy algebra*. Given a query Q we can generate a formula φ such that $\mathbf{P} \vdash \varphi$ if and only if $f(Q, \mathbf{P}) = 1$. For example, the query above would be associated with the policy formula $\texttt{V1} \vee \texttt{V2}$, indicating that a principal who is granted access to at least one of `V1` and `V2` should be authorized to execute the query. If the principal has access to `V2` but not `V1` then we can easily verify that $\texttt{V2} \vdash \texttt{V1} \vee \texttt{V2}$, which means the query is allowed. The RM uses such formulas, which we refer to as *policy formulas*, to make policy decisions as well as to generate explanations.

The contributions of this paper are as follows. First (Sections 3.1, 3.2) we provide a formal foundation for data-derived policies that connects disclosure lattices and policy algebras. Each query is associated with a policy formula which is automatically computed to reflect the true information disclosure by Q. To the best of our knowledge, our work is the first to discover and exploit the connection between policy algebras and disclosure lattices.

Second (Section 3.3), we introduce *explanations*, which are mathematical objects that the RM can return to the principal together with its decision. The explanation, which concisely represents the permissions that contributed to the RM's decision, gives principals whose queries were denied accurate guidance for reformulating their queries or requesting additional permissions.

Third, we show how our foundation can be used to build a security solution for real SQL (Section 4). We define a core language of *filter-project queries* that is simple enough to reason about in a mathematically sound manner but is expressive enough to capture a wide variety of real-world security policies. Next, we show how arbitrary SQL queries can be compiled to our core language. Due to the complexity of the SQL query language, the compilation process is necessarily conservative and may lead to unnecessary query denials. However, we can help developers address such denials by automatically identifying the parts of a query that were responsible for its rejection. We have fully implemented our solution; we explain the architecture and functionality of our prototype system.

Fourth, we present experimental results from an evaluation of our system on realistic workloads (Section 5).

2. THEORETICAL BACKGROUND

This section presents the necessary theoretical background for our work, including in particular disclosure lattices [8].

2.1 Query Languages

The results in Section 3 are independent of any specific database query language. However, it is useful to show examples in a concrete language to facilitate presentation. We therefore illustrate our results with conjunctive queries under bag semantics. We stress, however, that the same theoretical framework is applicable to other query languages as well, including conjunctive queries under set semantics [8] and *filter-project* queries (formally defined in Section 4.2), which combine elements of both set and bag semantics.

A conjunctive query (or view) 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. 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. Classically, conjunctive queries are evaluated under *set semantics*, which means that relations do not contain duplicates and duplicates are removed from the result of each query. However, we instead focus on conjunctive queries under *bag semantics*, where both relations and query answers can contain duplicates; bag semantics more accurately model the execution of real SQL queries. [13]

For the database schema from Figure 1, define

$$V_1(n, h) :- \texttt{U}(1, n, h)$$
$$V_2(n) :- \texttt{U}(1, n, h)$$
$$V_3(u, n) :- \texttt{U}(u, n, h), \texttt{F}(1, u)$$

where `U` and `F` refer to the `User` and `Friend` relations respectively. Under bag semantics, the views V_1, V_2, and V_3 correspond exactly to the views `V1`, `V2`, and `V3` in Figure 1. If the views were instead evaluated under set semantics then duplicate tuples would automatically be removed from the outputs of V_1, V_2, and V_3.

Just as in SQL, the output of a set of views $\mathbf{V}$ can be used as the input to a query Q. If every predicate referenced in the body of Q is a view in $\mathbf{V}$ then we say that Q is a *rewriting using* $\mathbf{V}$. The following is a rewriting using $\{V_1, V_3\}$:

$$Q_4(u) :- V_1(n, h), V_3(u, n)$$

The following queries are *not* rewritings using $\{V_1, V_3\}$. The first body atom of each query directly references the `User` relation rather than V_1 or V_3.

$$Q_5(u) :- \texttt{U}(1, n, h), V_3(u, n)$$
$$Q_6(u) :- \texttt{U}(1, n, h), \texttt{U}(u, n, h'), \texttt{F}(1, u)$$

However, we can show that Q_4 and Q_6 return the same answer on every possible dataset. Since the body of Q_4 just references views in $\{V_1, V_3\}$ and the body of Q_6 just references relations in the base schema, we say that Q_4 is an *equivalent rewriting of* Q_6 *using* $\{V_1, V_3\}$. The existence of such a rewriting implies that the answers to V_1 and V_3 uniquely determine the answer to Q_6 on any possible dataset.

2.2 Disclosure Orders

We now present *disclosure orders* [8], which formalize the intuition that some views (or sets of views) reveal more information about the dataset than others. We treat views as formal objects drawn from some finite universe $\mathcal{U}$. The relative amounts of information revealed by different sets of views can be compared using a binary relation $\preceq$ that operates on subsets of $\mathcal{U}$. Intuitively, if $\mathbf{V}$ and $\mathbf{V}'$ are sets of views, $\mathbf{V} \preceq \mathbf{V}'$ means that the answers to all the views in $\mathbf{V}$ are uniquely determined by the views in $\mathbf{V}'$.

Definition 1. *A* ***disclosure order*** *is a binary relation on subsets of* $\mathcal{U}$ *such that*

(i) If $\mathbf{V} \subseteq \mathbf{V}'$ *then* $\mathbf{V} \preceq \mathbf{V}'$.

(ii) If $\mathbf{V} \preceq \mathbf{V}'$ *and* $\mathbf{V}' \preceq \mathbf{V}''$ *then* $\mathbf{V} \preceq \mathbf{V}''$.

(iii) If $\mathbf{V} \preceq \mathbf{V}''$ *and* $\mathbf{V}' \preceq \mathbf{V}''$ *then* $\mathbf{V} \cup \mathbf{V}' \preceq \mathbf{V}'$.

The first axiom ensures that adding new elements to a set of views can only increase the amount of information it

reveals about the dataset. The second axiom is a standard transitivity condition. The third axiom ensures that upper bounds on information disclosure remain meaningful even when information is combined across multiple queries.

We next present three different examples of disclosure orders. In each case, we assume that $\mathcal{U}$ is the set of conjunctive queries under bag semantics.[1]

(i) **Set containment**: $\mathbf{V} \preceq \mathbf{V}'$ if and only if $\mathbf{V} \subseteq \mathbf{V}'$.

(ii) **View determinacy**: $\mathbf{V} \preceq \mathbf{V}'$ if and only if the answers to the views in $\mathbf{V}'$ uniquely determine the answers to the views in $\mathbf{V}$ on every possible dataset.

(iii) **View rewriting**: $\mathbf{V} \preceq \mathbf{V}'$ if and only if every view in $\mathbf{V}$ has a rewriting using $\mathbf{V}'$.

Although every disclosure order is a *preorder* (i.e., it is reflexive and transitive), disclosure orders are not generally partial orders. The reason is that two different views can reveal the same information. For instance, define

$$V_7(u_1, u_2) :- \mathtt{F}(u_1, u_2)$$
$$V_8(u_2, u_1) :- \mathtt{F}(u_1, u_2)$$

Intuitively, the views should reveal the same information about the dataset despite being unequal because the answer to each view can be computed from the other.

We are not aware of any efficient solution for checking dataset-independent view determinacy; under slightly different assumptions, it is an open question whether the problem is even decidable. [19] In contrast, checking set containment is trivial and checking whether an equivalent rewriting exists is NP-complete in the size of the input. [13] Since queries are typically quite small, this check is fast in practice.

2.3 Disclosure Lattices

Disclosure orders can be extended to *disclosure lattices*, which allow us to answer two basic questions. First, given two sets of views $\mathbf{V}$ and $\mathbf{V}'$, what *common* information is revealed both by $\mathbf{V}$ alone and by $\mathbf{V}'$ alone? And second, what *combined* information can be learned from looking at the answers to the views in $\mathbf{V}$ and $\mathbf{V}'$ together? By combining two sets of views, it is often possible to obtain information that cannot be learned by looking at either set on its own; this is the classic problem of *information combination* [18].

We would like sets of views to form a lattice where the *greatest lower bound* (GLB) of $\mathbf{V}$ and $\mathbf{V}'$ (denoted $\mathbf{V} \sqcap \mathbf{V}'$) represents the information that is common to the two sets and the *least upper bound* (LUB) of $\mathbf{V}$ and $\mathbf{V}'$ (denoted $\mathbf{V} \sqcup \mathbf{V}'$) represents the combined information from the two sets. However, this is not quite right: if $\preceq$ was a lattice order then it would need to be antisymmetric, so that $\mathbf{V} \preceq \mathbf{V}'$ and $\mathbf{V}' \preceq \mathbf{V}$ would together imply $\mathbf{V} = \mathbf{V}'$. We have seen that disclosure orders need not satisfy this property.

To fix the problem, we define a new unary operator $(\Downarrow \mathbf{V})$:

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

Intuitively, $(\Downarrow \mathbf{V})$ contains every view in $\mathcal{U}$ whose answer is uniquely determined by $\mathbf{V}$. We can use this new operator to obtain a *disclosure lattice* with the desired properties. [8]

[1] To ensure that $\mathcal{U}$ is finite, we assume that queries are expressed in a language with finitely many distinct symbols and that the size of every query is bounded above by a very large but arbitrarily selected constant.

$$V_9(u, n, h) :- \mathtt{U}(u, n, h)$$
$$V_{10}(u, n) :- \mathtt{U}(u, n, h)$$
$$V_{11}(1, n, h) :- \mathtt{U}(1, n, h)$$
$$V_{12}(h) :- \mathtt{U}(u, n, h)$$
$$V_{13}(u_1, u_2) :- \mathtt{F}(u_1, u_2)$$
$$V_{14}(1, u_2) :- \mathtt{F}(1, u_2)$$

Figure 3: A set of security views

THEOREM 1. *Let $\mathcal{U}$ be a set of views, and let $\preceq$ be a disclosure order for $\mathcal{U}$. Define $\mathcal{I} = \{\Downarrow \mathbf{V} : \mathbf{V} \subseteq \mathcal{U}\}$. Then $(\mathcal{I}, \preceq)$ is a lattice; details are as follows:*

- *LUB:* $(\Downarrow \mathbf{V}) \sqcup (\Downarrow \mathbf{V}') = \Downarrow (\mathbf{V} \cup \mathbf{V}')$.
- *GLB:* $(\Downarrow \mathbf{V}) \sqcap (\Downarrow \mathbf{V}') = (\Downarrow \mathbf{V}) \cap (\Downarrow \mathbf{V}')$.
- *Top element* $\top = (\Downarrow \mathcal{U}) = \mathcal{U}$*, bottom element* $\bot = (\Downarrow \emptyset)$.

Data-derived policy functions can be expressed in terms of disclosure lattices. Suppose a principal who is authorized to see the contents of a set of security views $\mathbf{P}$ tries to execute a query Q. The query is permitted only when the views in $\mathbf{P}$ contain enough information to answer Q, which is true precisely when $\{Q\} \preceq \mathbf{P}$. Translated into the lattice above, the check is equivalent to one that requires Q to be answerable using the combined information from all the views in $\mathbf{P}$. Equivalently, we require that $\Downarrow \{Q\} \preceq \Downarrow \mathbf{P}$.

3. POLICIES AND EXPLANATIONS

Now that we have introduced disclosure lattices, we can move to the first two major contributions of our paper. First, we introduce *policy formulas* that explicitly represent which combinations of views a principal must have access to before executing a given query. We explain how to compute policy formulas that are *data-derived*. Next, we show how to use policy formulas to generate data-derived *explanations* for system decisions to allow or deny queries.

3.1 Policy Formulas

We start with a set of *security views* $\mathbf{V}$ defined by a human administrator. Each view in $\mathbf{V}$ roughly corresponds to a single *permission* that a principal may or may not be granted. A policy formula is a Boolean formula in which every security view is treated as a variable. *Policy formulas over* $\mathbf{V}$ are formally defined using the BNF grammar

$$\tau \quad ::= \quad 0 \quad | \quad 1 \quad | \quad V \quad | \quad (\tau \vee \tau) \quad | \quad (\tau \wedge \tau)$$

where each V is a security view in $\mathbf{V}$. The constant 0 corresponds to a policy in which a query's execution is never permitted, while the constant 1 corresponds to a policy in which a query's execution is always permitted.

Consider the set of security views in Figure 3. The policy formula V_9 indicates that a principal must be granted access to the view V_9 in order to execute a given query. The formula $(V_9 \vee V_{11})$ indicates that the principal must be granted access to *either* the view V_9 *or* the view V_{11}. The formula $(V_9 \wedge V_{14})$ indicates that the principal must be granted access to *both* the view V_9 *and* the view V_{14}.

In general, a policy formula φ can be attached to any query Q (or more generally, any *set* of queries) over the

dataset; the formula is then used by the reference monitor to decide whether a given principal can execute Q. Given the set of security views $\mathbf{P}$ that the principal is granted access to, the reference monitor (RM) checks whether $\mathbf{P} \vdash \varphi$. If so, the RM will allow the principal to execute Q; otherwise, the RM will deny the request. This check can be performed in linear time by recursively applying the following rules:

$$\mathbf{P} \vdash 1 \text{ and } \mathbf{P} \nvdash 0$$
$$\mathbf{P} \vdash V \text{ if and only if } V \in \mathbf{P}$$
$$\mathbf{P} \vdash (\varphi \vee \psi) \text{ if and only if } \mathbf{P} \vdash \varphi \text{ or } \mathbf{P} \vdash \psi$$
$$\mathbf{P} \vdash (\varphi \wedge \psi) \text{ if and only if } \mathbf{P} \vdash \varphi \text{ and } \mathbf{P} \vdash \psi$$

It is important to realize that the check described above is agnostic to the question of how the views in $\mathbf{P}$ are computed. Consequently, the model is relevant even for systems that rely on role-based access control or similar schemes where the process of computing $\mathbf{P}$ can be arbitrarily complicated.

3.2 Generating Data-Derived Formulas

We now show how to generate data-derived policy formulas for database queries. Recall that a data-derived policy specifies that a principal with access to views in $\mathbf{P}$ can receive the answer to query Q if and only if $\{Q\} \preceq \mathbf{P}$. As explained in the Introduction, most policies used in practice are not data-derived; we believe this is because checking directly whether $\{Q\} \preceq \mathbf{P}$ makes policy decisions computationally expensive and tricky for humans to understand.

On the other hand, given a policy formula φ, checking whether $\mathbf{P} \vdash \varphi$ is straightforward and can be performed quickly even for large formulas. We therefore propose to encode data-derived policy functions using policy formulas. That is, we want a system that can take a query Q and compute a policy formula φ such that, for any subset $\mathbf{P}$ of the security views, $\mathbf{P} \vdash \varphi$ if and only if $\{Q\} \preceq \mathbf{P}$. Intuitively, φ tells us precisely which combinations of the security views contain enough information to uniquely determine the answer to Q. For example, consider the query

$$Q_{15}(h) :- \mathtt{U}(1, n, h)$$

Given the security views defined in Figure 3, Q_{15} can be answered if the principal is granted access to at least one of the views V_9 or V_{11}. This is both a necessary and a sufficient condition. No other view except V_{12} reveals the Hobby field of the `User` relation, and V_{12} does not contain enough information to determine which hobby is associated with User 1. The policy formula for Q_{15} is therefore

$$(V_9 \vee V_{11})$$

The question arises whether it is always possible to compute the formula φ we desire. The answer is *yes*; there is a very deep connection between disclosure lattices and policy formulas. Given any set of queries $\mathbf{Q}$ over the dataset there is a policy formula φ such that $\mathbf{P} \vdash \varphi$ if and only if $\mathbf{Q} \preceq \mathbf{P}$ for every subset $\mathbf{P}$ of the security views. Furthermore, it is possible to compute φ in a completely automated fashion.

Efficiency, however, is another matter entirely. The number of security views can be quite large, and the size of a policy formula can be exponential in the number of security views. We must limit policy formulas to a manageable size if they are to be used in practice. We can ensure that this is the case by requiring the universe $\mathcal{U}$ of possible views to be *decomposable*.

```
1: procedure POLICY(Q, V)
2:     φ ← 1
3:     for Q ∈ Q do
4:         ψ ← 0
5:         for V ∈ V do
6:             if {Q} ⪯ {V} then
7:                 ψ ← (ψ ∨ V)
8:             end if
9:         end for
10:        φ ← (φ ∧ ψ)
11:    end for
12:    return φ
13: end procedure
```

Figure 4: Algorithm for computing data-derived policy formulas (decomposable case)

Formally, $\mathcal{U}$ is said to be *decomposable* if for every query Q and all sets of views $\mathbf{V}$ and $\mathbf{V}'$, $\{Q\} \preceq \mathbf{V} \cup \mathbf{V}'$ implies $\{Q\} \preceq \mathbf{V}$ or $\{Q\} \preceq \mathbf{V}'$. Put another way, if neither $\mathbf{V}$ alone nor $\mathbf{V}'$ alone contains enough information to answer Q then the union of the two sets still will not contain enough information to answer Q.

Decomposability holds when we restrict our attention to the universe $\mathcal{U}_{single}$ of single-atom conjunctive queries and views; it does not generally hold when applied to multi-atom queries and views in the obvious way. We restrict our attention to single-atom queries and views for now and explain how to handle a more general class of queries and views in Section 4.2.

When decomposability holds, it is possible to compute concise policy formulas for arbitrary queries. Given a set of queries $\mathbf{Q}$ and a set of security views $\mathbf{V}$, the following formula allows us to compute the formula governing $\mathbf{Q}$:

$$\text{policy}(\mathbf{Q}) = \bigwedge_{Q \in \mathbf{Q}} \left(\bigvee_{V \in \mathbf{V}:\ \{Q\} \preceq \{V\}} V \right)$$

For any subset $\mathbf{P}$ of the security views, $\mathbf{P} \vdash \text{policy}(\mathbf{Q})$ if and only if $\mathbf{Q} \preceq \mathbf{P}$. Consequently, policy($\mathbf{Q}$) tells us exactly which combinations of the security views reveal enough information to answer all the queries in $\mathbf{Q}$. Furthermore, its size is polynomial in both the number of queries and the number of security views.

The algorithm in Figure 4 computes the policy formula for a set of queries $\mathbf{Q}$ with respect to a collection of security views $\mathbf{V}$. It begins by setting the policy formula to 1 (Line 2) – an empty set of queries should always be permitted. It then loops through the list of queries in $\mathbf{Q}$ (Lines 3-11), computes a separate policy formula for each query (Line 4-9), and returns the conjunction of all the resulting formulas (Line 12); the result is a policy formula over the views in $\mathbf{V}$.

The inner loop (Lines 4-9) computes the policy formula for a single query Q. It starts by setting the formula to 0 (Line 4) – if no security view contains enough information to answer Q then the query's execution should be disallowed. It then loops through all the security views in $\mathbf{V}$ and takes the disjunction of all the views that contain enough information to answer Q (Lines 5-9).

For example, let $\mathbf{Q}$ contain the queries

$$Q_{16}(1, n) :- \mathtt{U}(1, n, h)$$
$$Q_{17}(h) :- \mathtt{U}(u, n, h)$$

and let $\mathbf{V}$ consist of the views from Figure 3. Then

$$\text{policy}(\{Q_{16}\}) = V_9 \vee V_{10} \vee V_{11}$$
$$\text{policy}(\{Q_{17}\}) = V_9 \vee V_{12}$$

and therefore

$$\text{policy}(\{Q_{16}, Q_{17}\}) = (V_9 \vee V_{10} \vee V_{11}) \wedge (V_9 \vee V_{12})$$

For the other direction, given a policy formula φ, the recursive function $\text{disc}(\varphi)$ defined below allows us to determine precisely what information about the dataset (represented as a point in the disclosure lattice $\mathcal{I}$) can be obtained by executing only queries that are permitted by φ:

$$\begin{aligned} \text{disc}(\varphi \vee \psi) &= \text{disc}(\varphi) \sqcap \text{disc}(\psi) \\ \text{disc}(\varphi \wedge \psi) &= \text{disc}(\varphi) \sqcup \text{disc}(\psi) \\ \text{disc}(V) &= \Downarrow V \end{aligned}$$

In the example above,

$$\begin{aligned} &\text{disc}((V_9 \vee V_{10} \vee V_{11}) \wedge (V_9 \vee V_{12})) \\ &= (\Downarrow V_9 \sqcap \Downarrow V_{10} \sqcap \Downarrow V_{11}) \sqcup (\Downarrow V_9 \sqcap \Downarrow V_{12}) \end{aligned}$$

$\mathbf{P} \vdash \varphi$ implies $\text{disc}(\varphi) \preceq \mathbf{P}$, but the converse does not always hold because $\mathbf{P}$ can be inconsistent. For instance, it is possible to grant a principal access to V_9 but not V_{10} even though the former reveals strictly more information about the dataset than the latter. If we disallow such inconsistencies, the reverse implication holds as expected.

In summary, we have shown how to represent the security policy governing any query over the dataset as a data-derived policy formula. We do this by exploiting a deep connection between disclosure lattices that precisely quantify information disclosure and policy formulas that are easy for a reference monitor to evaluate. For every query Q we compute a formula φ; the reference monitor makes policy decisions by evaluating φ against the list of permissions $\mathbf{P}$ that were granted to the principal. If the universe of queries is decomposable then policy formulas are guaranteed to be small and easy to reason about. However, it is possible to compute policy formulas for queries even when decomposability does not hold.

3.3 Explanations

Policy formulas are not just useful for enforcement; they can be used to generate *explanations*, which are mathematical objects that justify the RM's policy decisions.

Explanations come in two varieties: *why-so explanations* and *why-not explanations*. If the execution of a query Q is authorized, a why-so explanation indicates which of the principal's permissions were responsible for the positive authorization decision. If the execution of Q is not authorized, a why-not explanation indicates which additional permissions need to be granted before the query can be successfully executed. Both types of explanations can be efficiently computed from $\text{policy}(Q)$.

As before, let $\mathbf{Q}$ denote a set of queries, $\mathbf{V}$ denote a set of security views, and let $\mathbf{P} \subseteq \mathbf{V}$ be the subset of the security views that a principal has been granted access to. A why-so explanation for Q characterizes exactly which subsets of $\mathbf{P}$ contain enough information to answer $\mathbf{Q}$. It can be obtained by replacing every view V in $\text{policy}(\mathbf{Q})$ that the principal has not been granted access to with the constant 0. For example, suppose $\mathbf{Q} = \{Q_{16}, Q_{17}\}$, so that

$$\text{policy}(\mathbf{Q}) = (V_9 \vee V_{10} \vee V_{11}) \wedge (V_9 \vee V_{12})$$

If the principal is granted access to V_9 and V_{10} then the why-so explanation for $\mathbf{Q}$ w.r.t. $\{V_9, V_{10}\}$ is

$$(V_9 \vee V_{10} \vee 0) \wedge (V_9 \vee 0) = (V_9 \vee V_{10}) \wedge (V_9) = V_9$$

indicating that V_9 was solely responsible for the positive authorization decision. If the principal is instead granted access to $\{V_{10}, V_{11}, V_{12}\}$, the why-so explanation for $\mathbf{Q}$ is

$$(0 \vee V_{10} \vee V_{11}) \wedge (0 \vee V_{12}) = (V_{10} \vee V_{11}) \wedge (V_{12})$$

indicating that in order to answer the query, the principal would need access to V_{12} and at least one of V_{10} and V_{11}.

If the RM's authorization decision is negative, a why-not explanation identifies the additional permissions that would need to be granted in order to reverse the decision. A why-not explanation is obtained by replacing every view V in $\text{policy}(\mathbf{Q})$ that the principal *has* been granted access to with the constant 1. If

$$\text{policy}(\mathbf{Q}) = (V_9 \vee V_{10} \vee V_{11}) \wedge (V_9 \vee V_{12})$$

then the why-not explanation for $\mathbf{Q}$ w.r.t. $\{V_{10}, V_{11}\}$ is

$$(V_9 \vee 1 \vee 1) \wedge (V_9 \vee V_{12}) = 1 \wedge (V_9 \vee V_{12}) = (V_9 \vee V_{12})$$

indicating that in order to execute the queries in $\mathbf{Q}$, the principal would need to be granted additional access to at least one of the permissions V_9 and V_{12}. Similarly, the why-not explanation for $\mathbf{Q}$ w.r.t. $\{V_{12}\}$ is

$$\begin{aligned} & (V_9 \vee V_{10} \vee V_{11}) \wedge (V_9 \vee 1) \\ = & (V_9 \vee V_{10} \vee V_{11}) \wedge 1 \\ = & (V_9 \vee V_{10} \vee V_{11}) \end{aligned}$$

indicating that the principal would need to be granted access to at least one of the views V_9, V_{10}, and V_{11} before all the queries in $\mathbf{Q}$ could be answered.

Why-so and why-not explanations are helpful to both the principal and the system administrator. From the administrator's perspective, if the principal holds some permissions that never show up in why-so explanations for his or her queries, this may be an indication that the principal is *over-privileged* and that those unused permissions should be revoked. On the other hand, from the principal's perspective, if a query is denied, the why-not explanation provides clear guidance on which additional permissions to request from the appropriate granting entity. With our data-derived policy formulas, both kinds of explanations are straightforward to compute at query time.

4. DISCLOSURE CONTROL IN PRACTICE

At this point we have given a theoretical foundation for computing data-derived policy formulas and explanations. We now introduce our system, which generates policy formulas and explanations for real SQL queries. In addition, our system can handle realistic security policies that are *reflective*, which means that the policies governing tuples in one part of the database can depend on a different part of the same database. Such policies arise frequently in practice. For instance, Facebook's security policies distinguish

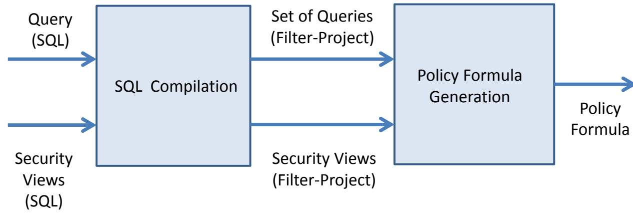

Figure 5: Computing policy formulas – System Architecture

between queries that reveal information about friends of the current user and queries that reveal information about arbitrary users; the former may be permitted when the latter are disallowed. To express and enforce such a policy, the following security view is required:

```
SELECT U1.uid, U1.name FROM User U1 WHERE U1.uid IN
  (SELECT F1.uid2 FROM Friend F1 WHERE F1.uid1 = 1)
```

This security view contains a semijoin and as such cannot be handled efficiently by algorithms in previous work [8].

4.1 Overview

Our prototype system can handle a wide variety of features that arise in real SQL queries, including:

- Aggregate operators such as `SUM` and `COUNT`.
- Algebraic expressions, equalities, and inequalities.
- Logical boolean connectives, including `AND`, `OR`, and `NOT`.
- Nested subqueries including the `EXISTS` and `IN` keyword and set-comparisons operators such as `X >= ANY (...)`. We also support correlated subqueries.
- `JOIN ON` and `WHERE` clauses.
- `GROUP BY` and `HAVING` clauses.
- Left, right, and full outer joins.
- Null values, `IS NULL`, and `IS NOT NULL`.
- The `UNION`, `INTERSECT`, and `EXCEPT` set operators.
- Temporary tables.

If additional SQL features are desired, it is possible to extend our implementation to support them.

Security views can also be defined using standard SQL syntax, but not all SQL features are supported. Every security view must be expressible as a *filter-project* query. Roughly speaking, this means that each security view must be a `SELECT-FROM-WHERE` query with semijoins but not inner or outer joins. While this is a nontrivial restriction, it is sufficient to capture many security constraints that arise in practice, including the Facebook friends query above.

Filter-project queries represent a sweet spot, as they are simple enough to reason about in a mathematically rigorous way but sophisticated enough to capture a wide range of real-world security policies.

The high-level architecture of our system is shown in Figure 5. Computing policy formulas is a two-stage process. First, queries issued by a principal are fed to a SQL processor that compiles them into filter-project queries. Compilation is conservative: the compiled versions of the queries disclose at least as much information as the original SQL queries, and may reveal strictly more information in some cases. This may cause the reference monitor to overestimate the information disclosed by and consequently reject certain legal queries. It will, however, ensure that illegal queries are never accepted. A policy formula therefore represents an upper bound on the information needed to answer a query.

Security views are similarly compiled into filter-project queries at initialization time; the compilation process for security views is lossless due to the restriction we imposed above. Once the compilation is complete, the system computes a policy formula for the compiled query using the algorithm from Figure 4. The entire pipeline is database-independent; policy formulas and explanations can reveal information about the database schema but cannot leak information about the contents of the underlying database.

The remainder of this Section describes the architecture of our system in more detail. In Section 4.2 we formally introduce filter-project queries and describe how to compute policy formulas and explanations for these queries. In Section 4.3 we discuss the early stages of our compilation pipeline, which output a collection of single-atom projection queries without equality constraints. In Section 4.4 we discuss the later stages of our compilation pipeline, which convert the single-atom projection queries to filter-project queries.

4.2 Filter-Project Queries

We now motivate and formalize filter-project queries, which serve as an intermediate representation within our system.

We begin by introducing some terminology, namely *tables* and *table instances*. *Tables* are query-independent and belong to the underlying database. *Table instances*, on the other hand, originate in the `FROM` clauses of SQL queries. Thus, a query containing a self-join has two table instances yet references just one table. The query

```
SELECT U1.uid, U1.name FROM User U1, Friend F1
 WHERE F1.uid1 = 1 AND F1.uid2 = U1.uid
```

has two table instances: `U1` and `F1`.

The reference monitor's job is determine whether the execution of a specified query should be permitted or denied. In our system, the RM ends up performing a separate policy check for each table instance in the query. This policy check takes into account two different types of information. First, which *columns* in each table instance can affect the query's output? And second, which *rows* in each table instance can affect the query's output?

The first question can be answered relatively easily. In the query above, the columns `uid` and `name` of the table instance `U1` can affect the query's output, and the columns `uid1` and `uid2` of the table instance `F1` can affect the query's output.

The answer to the second question is more subtle. A row in `F1` can affect the query's output if (i) `F1.uid1` is equal to 1 and (ii) there is a row in the `User` relation whose `uid` is equal to `F1.uid2`. Similarly, a row in `U1` can affect the query's output if there is a corresponding row in `Friend` whose `uid1` is equal to 1 and whose `uid2` is equal to `U1.uid`.

In order to encode the types of constraints described above, we define a class of queries called *filter-project queries*. These are related to the familiar select-project and select-project-join queries, but they are distinct from both.

A filter-project query contains a *head atom* H, a *main body atom* B (or just a *body atom* for short), and zero or more *filter atoms* $F_1, F_2, \ldots$, and is written as

$$H :- B \ltimes F_1 \wedge F_2 \wedge \ldots$$

When the set of filter atoms is empty, this is shortened to

$$H :- B$$

As the latter syntax implies, filter-project queries generalize single-atom conjunctive queries under bag semantics.

In our running example, the filter-project query for U1 is

$$Q_{18}(u, n) :- \texttt{U}(u, n, h) \ltimes \texttt{F}(1, u)$$

This encodes both the row-based constraints and the column-based constraints mentioned above. The distinguished variables in the query's head are bound to the columns `uid` and `name` of the `User` relation. And the tuples that appear in the query's output are precisely those for which U1 is joined with a tuple of the `Friend` relation that satisfies `uid1 = 1`.

Similarly, the filter-project query associated with F1 is

$$Q_{19}(u) :- \texttt{F}(1, u) \ltimes \texttt{U}(u, n, h)$$

Notice that the answers to Q_{18} and Q_{19} jointly contain enough information to uniquely determine the answer to the original SQL query on any possible dataset.

The main atom of a filter-project query is evaluated under bag semantics, whereas the remaining atoms are evaluated under set semantics. The filter atoms of a query act as selection predicates in the sense that they can remove tuples from the query answer but cannot add new tuples. Furthermore, only variables that appear in the query's main body atom can be referenced in the query head.

In our running example, Q_{18} is equivalent to the following SQL query:

```
SELECT U1.uid, U1.name FROM User U1 WHERE U1.uid IN
  (SELECT uid2 FROM Friend WHERE uid1 = 1)
```

and Q_{19} is equivalent to

```
SELECT F1.uid2 FROM Friend F1 WHERE F1.uid1 = 1
   AND F1.uid2 IN (SELECT uid FROM User)
```

More generally, filter atoms in our query language correspond to semijoins in standard SQL.

We can now define a *disclosure order* (Section 2.2) that allows us to compare the information revealed by different filter-project queries. In contrast with the disclosure orders seen earlier, we order filter-project queries according to the amount of information that they disclose about their *main body atoms*. According to our criterion, the following query reveals strictly more information than Q_{18} above:

$$Q_{20}(u, n) :- \texttt{U}(u, n, h)$$

The reason is that Q_{18} just reveals information about `Users` who are friends of User 1, whereas Q_{20} reveals information about every tuple in the `User` relation. The analysis depends only on the query definitions and is independent of the contents of the underlying dataset.

Restricting our attention to the information disclosed by queries' main body atoms serves two purposes. First, it ensures that the RM will perform exactly one policy check for each table instance in the original SQL query. For the SQL query above, the RM will perform one check to determine whether the execution of Q_{18} is authorized and another to determine whether Q_{19} is authorized. If both are authorized then the SQL query will be accepted. Otherwise, the RM will be able to point to a specific table instance that was responsible for the failed authorization decision: U1 if the check for Q_{18} fails and F1 if the check for Q_{19} fails.

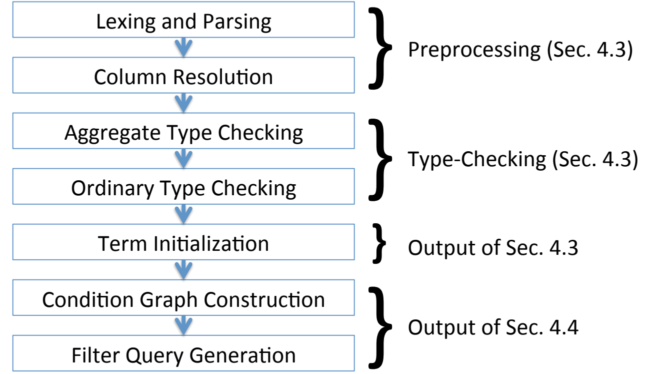

Figure 6: Query compilation pipeline

A second advantage of this disclosure order is that it ensures that the universe of filter-project queries is decomposable even if the queries contain multiple filter atoms. Consequently, filter-project queries are compatible with the efficient policy formula generation and explanation algorithms from Section 3.

We can formalize our desired disclosure order using *query homomorphisms* [11, 13, 14]. In our disclosure order, it is the case that $\{Q_{18}\} \preceq \{Q_{20}\}$. Given filter-project queries Q and Q', the problem of determining whether $\{Q\} \preceq \{Q'\}$ is NP-hard; however, queries are typically small in practice. Given sets of queries $\mathbf{Q}$ and $\mathbf{Q}'$, we can check whether $\mathbf{Q} \preceq \mathbf{Q}'$ using $O(|\mathbf{Q}| \cdot |\mathbf{Q}'|)$ comparisons between pairs of singleton queries; details are omitted due to space restrictions.

4.3 Preprocessing and Term Initialization

We now explain the early passes of our compiler, which transform an arbitrary SQL query into a set of *projection queries*, or single-atom conjunctive queries without constants or equality constraints. Although not particularly useful on its own, the pipeline will be extended in Section 4.4 in order to generate a collection of *filter-project* queries in the language described in Section 4.2

Our compiler begins by lexing and parsing the raw SQL queries using JSqlParser [1], an open-source query parsing library written in Java. We then execute a series of passes on the resulting *abstract syntax tree* (AST). Each pass corresponds to a complete traversal of the AST. A complete depiction of the passes is seen in Figure 6.

Our running example focuses on the query

```
SELECT name FROM User U1, Friend F1
 WHERE uid1 = 1 AND uid2 = uid
```

Preprocessing and column resolution: Parsing the raw SQL queries yields an AST representation as mentioned earlier. Once the AST is available, we identify table instances and perform column resolution. In our example, we identify U1 as a table instance associated with the `User` table and F1 as a table instance associated with the `Friend` table. Using information about the database schema, we infer that the columns `uid` and `name` belong to U1 whereas the columns `uid1` and `uid2` belong to F1. (See Figure 7a.) When applicable, column resolution also takes into account information about the query's nested structure.

```
(a) SELECT U1.name FROM User U1, Friend F1
     WHERE F1.uid1 = 1 AND F1.uid2 = U1.uid

(b) SELECT U1.name
      FROM User U1 LEFT OUTER JOIN Friend F1
        ON (F1.uid1 = 1 AND F1.uid2 = U1.uid)

(c) SELECT U1.name
      FROM User U1 FULL OUTER JOIN Friend F1
        ON (F1.uid1 = 1 AND F1.uid2 = U1.uid)

(d) SELECT U1.name
      FROM User U1, Friend F1, Friend F2
     WHERE F1.uid1 = 1 AND F1.uid2 = F2.uid1
       AND F2.uid2 = U.uid

(e) SELECT U1.name FROM User U1 WHERE U1.uid IN
      (SELECT F1.uid2 FROM Friend F1
        WHERE F1.uid1 = 1)
```

Figure 7: SQL queries handled by our system

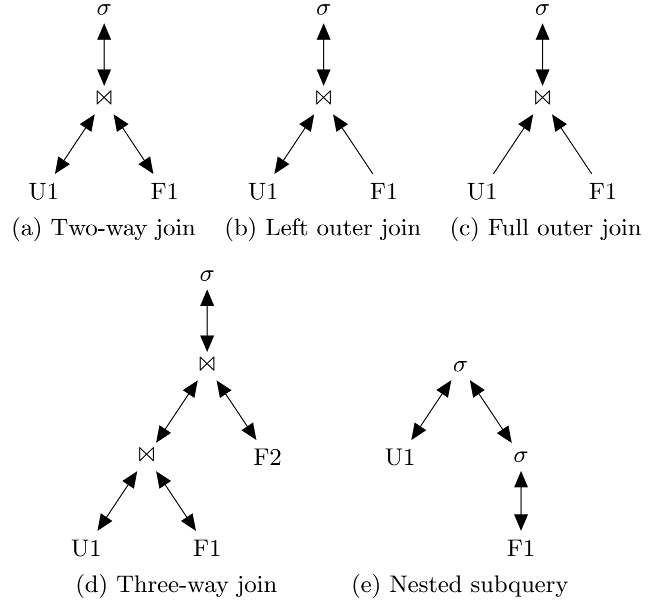

Figure 8: Condition graphs for queries from Fig. 7

Type-checking: We perform aggressive type-checking on the AST in order to ensure that it corresponds to a semantically well-defined query. The motivation for performing type-checking is simple: we can't reason about the disclosure of a query unless we understand its semantics. Our system performs two different kinds of type-checking. *Aggregate type-checking* detects bad queries such as

```
SELECT SUM(SUM(U1.uid)) FROM User U1
```

that mix aggregate and non-aggregate expressions in illegal ways. *Ordinary type-checking* detects bad queries such as

```
SELECT * FROM User U1 WHERE U1.uid IN
  (SELECT F1.uid1, F1.uid2 FROM Friend F1)
```

that provide invalid inputs to built-in functions, operators, and predicates.

Term initialization: The next step is to convert the AST for a SQL query Q into a collection of projection queries Q_1, Q_2, ..., Q_n that together reveal enough information to uniquely determine the answer to Q on any possible dataset. In Section 4.4 we will describe an extension to this process that converts the resulting projection queries into filter-project queries of the form described in Section 4.2.

The extraction proceeds as follows. First, we generate one query for each table instance in Q. For example, given any of the SQL queries from Figure 7a, Figure 7b, or Figure 7c, we generate two queries: $Q_{\texttt{U1}}$ and $Q_{\texttt{F1}}$. The sole body atom of $Q_{\texttt{U1}}$ references the `User` relation whereas the body atom of $Q_{\texttt{F1}}$ references the `Friend` relation.

We now determine which variables in our queries should be existential and which should be distinguished. A variable is marked as existential if the corresponding column is never referenced in the definition of Q, and is marked as distinguished in all other cases. This ensures that the output queries reveal all the columns needed to evaluate any arithmetic expression or inequality or logical predicate that appears in Q. We obtain the same values for $Q_{\texttt{U1}}$ and $Q_{\texttt{F1}}$ for all of the queries in Figures 7a, 7b, and 7c:

$$Q_{\texttt{U1}}(u, n) :- \texttt{U}(u, n, h)$$
$$Q_{\texttt{F1}}(u_1, u_2) :- \texttt{F}(u_1, u_2)$$

4.4 Generating Filter-Project Queries

The output of the compilation process described in Section 4.3 is a conservative upper bound on the information revealed by Q. However, it is missing information that is essential for driving policy decisions in practice. For instance, there is no way to know just by looking at $Q_{\texttt{F1}}$ that the query in Figure 7a only needs a list of friends of User 1 (Charles Babbage) rather than information about arbitrary tuples in the `Friend` relation. Similarly, there is no way to tell just by looking at $Q_{\texttt{U1}}$ that the query only needs to access tuples in the `User` relation that correspond to friends of User 1. In order to address these shortcomings, we build a *condition graph* that allows us to determine which equality constraints and filter atoms should be added to $Q_{\texttt{U1}}$ and $Q_{\texttt{F1}}$.

The condition graph for a query Q contains one node (labeled σ) for each `SELECT` statement in the query, one node (labeled $\bowtie$) for each two-way join in the query's logical evaluation plan, and one node for each table instance defined by the query. Figures 8a through 8e show the condition graphs for the SQL queries from Figures 7a through 7e respectively.

In many cases, condition graphs bear a deceptive resemblance to expression trees in the Relational Algebra. However, there are a number of important differences between the two. Edges in condition graphs can be unidirectional or bidirectional, and every condition graph must contain a `SELECT` node at its root. Furthermore, the same nodes appear in the graph regardless of whether we perform an inner join or an outer join. However, the directions of edges between those nodes can change depending on the type of join we are dealing with (Figures 8a, 8b, and 8c). And finally, `SELECT` nodes can have multiple children (Figure 8e). One child represents the `FROM` clause of the corresponding `SELECT` statement, while the remaining children represent correlated subqueries that appear in the statement's `WHERE` and `HAVING` clauses. Temporary tables (not shown here) are handled in a similar manner to nested subqueries.

Once we have constructed the condition graph, we can determine which semijoin filter atoms should appear in each

of the output queries. Our criterion is based on reachability in the condition graph: a directed path from table instance R to table instance S indicates that a tuple in R can only affect the query's output if it can be joined with a tuple from S. In our running example, $Q_{\texttt{U1}}$ should contain a filter atom for F1 precisely when the node associated with F1 is reachable from the node associated with U1 in the condition graph. For the query in Figure 7a we obtain

$$\begin{aligned} Q_{\texttt{U1}}(u,n) &:- \ \texttt{U}(u,n,h) \ltimes \texttt{F}(u_1,u_2) \\ Q_{\texttt{F1}}(u_1,u_2) &:- \ \texttt{F}(u_1,u_2) \ltimes \texttt{U}(u,n,h) \end{aligned}$$

The situation will be similar for the left outer join query in Figure 8b except that $Q_{\texttt{U1}}$ will not contain any filter atoms. For the full outer join query in Figure 8c neither $Q_{\texttt{U1}}$ nor $Q_{\texttt{F1}}$ will contain any filter atoms. An analogous criterion can be used for other types of queries, including those in Figures 8d and 8e from our running example.

The next step is to find equality constraints that should be added to $Q_{\texttt{U1}}$ and $Q_{\texttt{F1}}$. We begin by finding all SELECT nodes that are reachable from the condition graph node associated with U1. We then check the WHERE and HAVING clauses of the corresponding SELECT statements for equality constraints that must hold for all tuples in the statements' outputs. In the query from Figure 7a, we obtain two constraints: (F1.uid1 = 1) and (F1.uid2 = U1.uid). When we integrate both constraints into $Q_{\texttt{U1}}$ we obtain the query

$$Q_{\texttt{U1}}(u,n) \ :- \ \texttt{U}(u,n,h) \ltimes \texttt{F}(1,u)$$

An analogous process can be performed with $Q_{\texttt{F1}}$ to obtain

$$Q_{\texttt{F1}}(1,u_2) \ :- \ \texttt{F}(1,u_2) \ltimes \texttt{U}(u_2,n,h)$$

The filter-project queries $Q_{\texttt{U1}}$ and $Q_{\texttt{F1}}$ comprise the output of our extraction algorithm.

4.5 Summary and Extensions

Our system takes as input queries and security views expressed in SQL, with the restriction that security views must be expressible as filter-project queries. It then compiles each input SQL query to a set of filter-project queries. Because the universe of filter-project queries is decomposable, the system can generate policy formulas and explanations using the efficient algorithms from Section 3.

The number of SQL features that our system is able to handle has gradually increased throughout the course of its development and can be further increased as needed. By default, our system errs on the side of overestimating information disclosure when new language features are added. For example, an earlier version of our system required a principal who wished to execute the query

```
SELECT uid FROM User U1
 WHERE EXISTS (SELECT * FROM Friend F1)
```

to obtain permissions to access all of the columns in the Friend relation. The term initialization stage of our pipeline was subsequently extended with rules for handling selected columns in WHERE EXISTS subclauses more precisely.

5. EXPERIMENTAL EVALUATION

The goal of our experimental evaluation was three-fold. First, we wanted to verify that SQL queries containing a wide range of commonly used features could be correctly handled by the compiler discussed in Section 4. Second, we wanted to determine whether the language of filter-project queries described in Section 4.2 was powerful enough to represent a variety of practical security constraints. And third, we wanted to determine whether policy formulas could be generated quickly enough to be used in practical systems.

We implemented a prototype system in Java. Our system consisted of three main components: (i) a compiler for translating SQL queries into filter-project queries, (ii) a module for representing and reasoning about filter-project queries, and (iii) a module for representing and evaluating policy formulas and generating why-so and why-not explanations. The implementation of our compiler consisted of just over 5,500 lines of Java code, while filter-project queries were implemented in about 1,000 lines, and policy formulas were implemented in 250.

For the first goal, we began by writing end-to-end tests for our compiler based on several dozen SQL queries taken from a standard undergraduate database textbook. (See Chapter 5 of [21].) We gradually added more tests over time to keep track of the border cases we encountered. At the time of publication, we had 110 of our 115 end-to-end tests running successfully, and the remaining five were rejected as being semantically invalid. Of these, three were rejected due to limitations of JSqlParser. The last two were valid according to the SQL standard, but were considered to be invalid by both MySQL and PostgreSQL. The results gave us confidence that our compilation-based approach to disclosure control could be successfully applied to a broad range of real-world SQL features.

For the second and third goals, we ran experiments on a database inspired by the one used by the Facebook Query Language (FQL). [3] The schema has a rich and diverse set of security requirements. For example, albums containing media such as photos and videos can be made visible to friends of the current user, to friends of friends, or to the public at large. The membership of a group can be made world-visible, can be restricted to members of that group, or can be completely hidden. Lists of attendees for events can similarly be made world-visible can be restricted to other attendees. All of these contingencies were accounted for using filter-project security views.

Our test database schema contained 18 relations with a grand total of 269 distinct columns – or an average of 15 columns per relation. We deliberately selected relations that had nontrivial security policies. The number of columns per relation ranged from two for the Friend relation (which stores information about friendship relations between users) to 62 for the Application table (which stores information about third-party Facebook Apps). We ended up defining a total of 75 security views. Equality constraints appeared in all but one of our security views; the exception was a view over the Application table that listed certain types of publicly available information about Facebook Apps. Filter atoms appeared in 59 security views. Column projections were used in just four of the security views.

In addition to the qualitative experiments above, we also evaluated the performance of our prototype system. Our experiments were run on a workstation with two 4-core 2.13 GHz Intel Xeon processors and 50 GB of RAM. Our code was run on the OpenJDK 1.7 VM on Ubuntu 12.04.

Our experiments made use of a synthetic query generator that worked by recursively stringing together predefined templates. Our use of templates allowed us to gener-

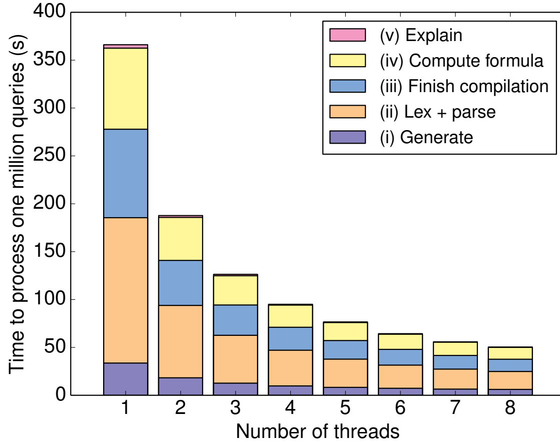

Figure 9: Performance for ordinary queries

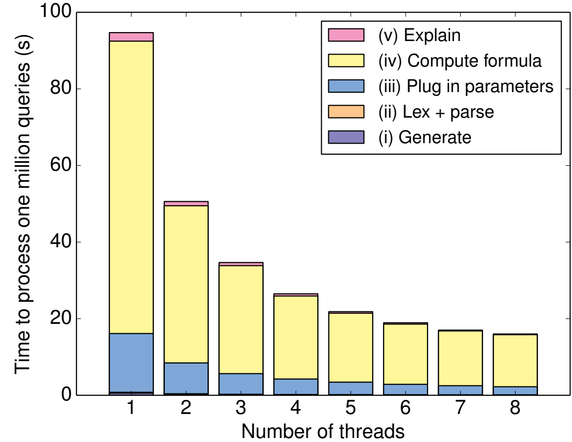

Figure 10: Performance for prepared statements

ate realistic-looking queries whose joins and equality predicates encoded meaningful security-related constraints. Each of the queries we generated contained between one and six table instances; the median 2 and the mean was 2.5. The experiment focused exclusively on filter-project queries with equality constraints. Such queries are especially challenging for our pipeline because they tend to yield filter-project queries with many filter atoms; experiments on other types of queries are available in a digital appendix.[2]

We measured the wall time needed to (i) randomly generate queries, (ii) lex and parse them, (iii) extract filter-project queries from the parsed output, (iv) compute policy formulas from the resulting filter-project queries, and (v) generate a policy decision for each query, a why-so explanation for each permitted query, and a why-not explanation for each denied query. Since each stage depended on the output of the previous stage, these measurements were cumulative: we measured the time for (i) alone, the time for (i) + (ii), and so on. We also varied the number of execution threads between 1 and 8. All of our algorithms are trivially parallelizable, and the threads operated completely independently of each other. In each case, we measured the time needed to process 1,000,000 queries. All numbers are averaged across five runs; the variation across runs was less than one second in almost all cases.

The results are shown in Figure 9. As expected, throughput scaled nearly linearly with the number of cores. The analysis pipeline took a total of 366 seconds on a single core or 50 seconds on 8 cores. If we exclude the time needed to randomly generate queries, these numbers change to 333 and 44 seconds respectively. If we focus on the last three stages (which are where we claim algorithmic contributions), the numbers fall to 181 and 26 seconds respectively. All computation was CPU-bound, and memory consumption ranged from 60 MB to 220 MB per thread. The time needed to perform policy checks and generate explanations was negligible compared to the rest of the system – about 3.5 seconds on a single core or 0.5 seconds on eight cores – and is barely visible in Figure 9.

[2] `http://www.cs.cornell.edu/~lucja/Publications/sigmod2014appendix.pdf`

Since query compilation time dominated the benchmark numbers in Figure 9, we speculated that throughput could be significantly improved by using prepared statements. The filter-project queries associated with prepared statements only needed to be calculated once (as a preprocessing step), and could retrieved on demand when the prepared statements were executed. Parameters passed to prepared statements at execution time could be substituted directly into the compiled filter-project queries, bypassing the need for a heavy-weight compilation step.

Results are shown in Figure 10. Our use of prepared statements yielded around a 3x improvement in overall throughput. The time spent on query generation, lexing and parsing was negligible, and the time needed to retrieve filter-project queries and plug in parameter values paled in comparison with the time needed to perform end-to-end query compilation. As a result, computation of policy formulas dominated the running time.

6. RELATED WORK

Disclosure lattices, which we originally proposed in [8], generalize the *lattice of information* [18]. Our current work is geared towards real-world SQL queries and complex security policies, while previous work focuses on conjunctive queries under set semantics and restricts security views to single-atom conjunctive queries.

Although there is a large body of work on enforcing access control policies through *view rewriting*, we are not aware of any previous work in the database security literature on generating human-readable explanations for policy decisions.

Access control mechanisms based on view rewriting can be classified as *Truman* or *non-Truman*. Non-Truman mechanisms (e.g., [22]) analyze queries to determine whether they should be accepted or rejected; the principal who issues a query receives either an answer (if the query is accepted) or a denial (if the query is refused). *Truman* mechanisms modify queries' execution semantics in order to ensure that they are consistent with an administrator-defined security policy; commercial Truman mechanisms include IBM LBAC [5] and Oracle VPD [6]. Our approach is non-Truman, but the use of filter-project security views is more commonly associated with Truman mechanisms (e.g., [12, 20]).

Our work has a strong connection to the problem of *query pricing* [17] proposed by Koutris et al. In the framework of query pricing, we are given a set of administrator-provided views (similar to our *security views*), and a monetary price is assigned to each view. Given a query Q, the goal of [17] is to find a subset of the views that contain enough information to answer Q and whose total cost is minimal subject to that constraint. Their model of information disclosure is database-dependent, whereas ours is database-independent. The policy formulas computed by our system can be used to derive linear programs that solve a *database-independent* variant of the query pricing problem. Details are omitted here due to space limitations.

Dwork's Differential Privacy [15] provides a formal basis for privacy-preserving data publishing. Differential privacy assigns a numeric privacy cost to any possible query over the dataset, but does not provide meaningful feedback about *whose* information is disclosed or *what types* of information are disclosed by a given query. In contrast, the policy formulas and explanations generated by our system are specifically designed to answer these types of questions.

In [10], Brodsky et al. propose an algorithm for enumerating the facts that could be learned from a collection of database queries. Their work provides a *lower bound* (expressed as a set of facts) on the information disclosed by a set of queries. Our techniques place an *upper bound* on the permissions needed to answer a set of queries.

Our *policy algebras* are related to formalisms proposed for composable security policies (e.g., [9]) and general-purpose authorization logics (e.g., NAL [24] and SecPal [7]). Compared to these systems, our model is simpler, easier to reason about, and can be used to produce data-derived policy formulas. Constructive authorization logics such as NAL ensure that all positive authorization decisions are accompanied by formal proofs that are in some sense analogous to our *why-so explanations*. We are not aware of any direct analogues to our *why-not explanations*.

7. FUTURE WORK

In the future, we plan to extend our system to make it even more broadly useful. Notably, we plan to explore the problem of generating *database-dependent* explanations that make use of information about the current state of the database. We also plan to improve the usefulness of the explanations by suggesting concrete possible courses of action for the principal and ranking them according to an appropriate metric.

7.1 Acknowledgments

This research was supported by 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. Any opinions, findings, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsors.

8. REFERENCES

[1] `https://github.com/JSQLParser/JSqlParser`.

[2] Android permissions. `http://developer.android.com/guide/topics/security/permissions.html`.

[3] Facebook query language. `https://developers.facebook.com/docs/reference/fql/`.

[4] Facebook query language (fql) reference. https://developers.facebook.com/docs/reference/fql/.

[5] Label-based access control (lbac) overview. `http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp?topic=%2Fcom.ibm.db2.udb.admin.doc%2Fdoc%2Fc0021114.htm`.

[6] Virtual private database. `http://www.oracle.com/technetwork/database/security/index-088277.html`.

[7] Moritz Y Becker, Cédric Fournet, and Andrew D Gordon. Secpal: Design and semantics of a decentralized authorization language. *CSF*, 2006.

[8] Gabriel M Bender, Lucja Kot, Johannes Gehrke, and Christoph Koch. Fine-grained disclosure control for app ecosystems. *SIGMOD*, 2013.

[9] Piero Bonatti, Sabrina De Capitani di Vimercati, and Pierangela Samarati. An algebra for composing access control policies. *TISSEC*, 2002.

[10] Alexander Brodsky, Csilla Farkas, and Sushil Jajodia. Secure databases: Constraints, inference channels, and monitoring disclosures. *TKDE*, 12(6):900–919, 2000.

[11] Ashok K Chandra and Philip M Merlin. Optimal implementation of conjunctive queries in relational data bases. *STOC*, 1977.

[12] Surajit Chaudhuri, Tanmoy Dutta, and S Sudarshan. Fine grained authorization through predicated grants. *ICDE*, 2007.

[13] Surajit Chaudhuri and Moshe Y Vardi. Optimization of real conjunctive queries. *SIGMOD*, 1993.

[14] Sara Cohen. Equivalence of queries that are sensitive to multiplicities. *VLDB*, 2009.

[15] Cynthia Dwork. Differential privacy. In *Automata, languages and programming*, pages 1–12. Springer, 2006.

[16] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. *CCS*, 2011.

[17] Paraschos Koutris, Prasang Upadhyaya, Magdalena Balazinska, Bill Howe, and Dan Suciu. Query-based data pricing. *PODS*, 2012.

[18] Jaisook Landauer and Timothy Redmond. A lattice of information. *CSFW*, 1993.

[19] Alan Nash, Luc Segoufin, and Victor Vianu. Views and queries: Determinacy and rewriting. *TODS*, 2010.

[20] Lars E Olson, Carl A Gunter, and P Madhusudan. A formal framework for reflective database access control policies. *CCS*, 2008.

[21] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003.

[22] Shariq Rizvi, Alberto Mendelzon, Sundararajarao Sudarshan, and Prasan Roy. Extending query rewriting techniques for fine-grained access control. *SIGMOD*, 2004.

[23] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[24] Fred B Schneider, Kevin Walsh, and Emin Gün Sirer. Nexus authorization logic (nal): Design rationale and applications. *TISSEC*, 14(1), 2011.