# Disjunctive Policies for Database-Backed Programs

Amir M. Ahmadian
*KTH Royal Institute of Technology*

Matvey Soloviev
*KTH Royal Institute of Technology*

Musard Balliu
*KTH Royal Institute of Technology*

***Abstract*—When specifying security policies for databases, it is often natural to formulate *disjunctive* dependencies, where a piece of information may depend on at most one of two dependencies $P_1$ or $P_2$, but not both. A formal semantic model of such disjunctive dependencies, the Quantale of Information, was recently introduced by Hunt and Sands as a generalization of the Lattice of Information. In this paper, we seek to contribute to the understanding of disjunctive dependencies in database-backed programs and introduce a practical framework to statically enforce disjunctive security policies. To that end, we introduce the *Determinacy Quantale*, a new query-based structure which captures the ordering of disjunctive information in databases. This structure can be understood as a query-based counterpart to the Quantale of Information. Based on this structure, we design a sound enforcement mechanism to check disjunctive policies for database-backed programs. This mechanism is based on a type-based analysis for a simple imperative language with database queries, which is precise enough to accommodate a variety of row- and column-level database policies flexibly while keeping track of disjunctions due to control flow. We validate our mechanism by implementing it in a tool, DIVERT, and demonstrate its feasibility on a number of use cases.**

## I. Introduction

Database security and information flow security have largely evolved as two disparate areas [1], [2], while sharing closely-related foundations and mechanisms to enforce security. Modern applications commonly rely on shared database backends to provide rich functionality to a multitude of mutually distrusting users. In response to frontend demands, database query languages, with features such as triggers, store procedures, and user-defined functions, have increasingly come to resemble full-fledged programming languages, thus calling into question the adequacy of the underlying access control models [3], [4]. A *security policy* describes the totality of expectations that we have of a computer system in the face of adversaries that seek to satisfy objectives that may differ from ours. In the context of database systems, whose purpose is to retain and provide information, the security policies of interest constrain who is allowed to learn what parts of that information. A class of such security policies which has proven particularly challenging to enforce with the methods of database security are *disjunctive policies*, which states that given two pieces of information, some entity may either learn one *or* the other, but not both.

A common example of disjunctive policies are databases which contain personally identifiable information, such as medical trial data. Biometric parameters of participants are important confounders that must be considered when drawing conclusions from the data, but at the same time releasing too many parameters of any one participant (such as their height, age and weight) might be sufficient to deanonymize them with high confidence [5]. Hence, a security policy for such a database may specify that the user may learn height and age, or height and weight, or age and weight, but not all three. Other examples of scenarios where disjunctive policies are useful include differential privacy [6] and secret sharing.

In this paper, we combine insights from database security and information flow research to develop a formal model for reasoning about disjunctive information in database-backed programs, and thus take a step towards reconciling the two fields. Our model makes it possible to reason about the semantic information dependencies in a program that performs queries, and compare them against a disjunctive policy. Building upon this, we propose a provably sound static enforcement mechanism that ensure that the policy is satisfied.

It is customary in information flow models to represent information as an equivalence relation on states, with the refinement order of equivalence relations corresponding to having more information. This representation can be used for both the actual information conveyed by a computational process and the bound imposed on it as part of a simple, non-disjunctive security policy. The possible equivalence relations on a given universe of states form a structure called the *Lattice of Information* (LoI) [7], in which security-relevant questions can be answered, such as whether a program reveals no more information than is allowed by the security policy, or what information is revealed by the combination of two programs. Similar questions have been addressed in the database community using an analogous object called the Disclosure Lattice [8]. We observe that this definition is actually insufficient to characterize information, which motivates us to introduce a more specific structure based on query determinacy, the *Determinacy Lattice* (DL). The formal relation between the Disclosure Lattice or our definition and LoI was hitherto unexplored, and more importantly neither of them can be used to represent disjunctions as seen in our motivating example.

Recently, Hunt and Sands [9] proposed a new information flow structure called the *Quantale of Information* (QoI), which seeks to address this shortcoming and establish a formal setting for representing, combining and comparing disjunctions of information. We build upon this work to introduce an analogous structure, the *Determinacy Quantale* (DQ), representing disjunctive dependencies in database-backed programs. As we show, this structure can be formally related to the QoI, and this relationship is analogous to that between the LoI and the DL. We then use the DQ to design a knowledge-based security

condition that relates disjunctive dependencies in database-backed programs to disjunctive policies.

We are the first to address the problem of enforcing disjunctive policies. Prior works that develop language-based enforcement techniques in database-backed applications do not support disjunctive policies, while database-level dependencies are restricted to coarse approximations that incorrectly reject secure programs, such as our previous example [10]–[14].

Perhaps unsurprisingly, path sensitivity of a static analysis is key to capturing disjunctive dependencies. We show how standard flow-sensitive type-based dependency analysis [15] can be adapted to a compositional path-sensitive analysis and thus capture disjunctive dependencies in terms of database queries. To represent these dependencies in the DQ model, we introduce a sound approximation of the information disclosed by each database query which is precise enough to represent complex combinations of both row- and column-level dependencies. Finally, in the DQ, the combination of these analyses can be proven sound with respect to our security condition. We expect that the overall architecture of the resulting soundness proof, in which we relate a sequence of abstractions of the behaviour of a program to ordered elements of the DQ, can be generalized to many other enforcement mechanisms for our security condition.

To demonstrate the practicality of our approach, we implement this type-based dependency analysis and query approximation for database-backed programs and evaluate it on a test suite and some use cases which effectively illustrate the need for disjunctive dependencies and disjunctive policies.

We refer the readers to the full version of the paper [16] for the proofs of the lemmas and theorems we present.

**Summary of contributions.**

- We introduce a formal model for reasoning about disjunctive dependencies and policies in databases. In the process, we show how to reconcile perspectives from the database security and information flow communities.
- We introduce a database-specific model of knowledge, the Determinacy Lattice, and a disjunctive extension, called the Determinacy Quantale, and explore their relationship to established general-purpose semantic models.
- Using our model, we define an extensional security condition for database-backed programs that accommodates disjunctive policies.
- We propose a type-based program analysis to capture disjunctive dependencies in database-backed programs, combine them with a novel abstraction of queries, and prove them sound with respect to our security condition. This is presented as an instance of a generalizable architecture for such soundness proofs.
- We implement a prototype tool that uses type-based dependency analysis and query approximation to verify query-based disjunctive policies for database-backed programs, and demonstrate its feasibility on a test suite and a number of use cases.

The rest of paper is structured as follows. After reviewing preliminaries in Section II, we give our account of the DL and introduce the DQ in Section III-C. In Section IV-B, we formalize our model of database-backed programs and the security policies we impose on them, culminating in a formal security condition. We present enforcement mechanisms in Section V, and their implementation and evaluation in Section VI. In Section VII, we contextualize our contributions with a discussion of related work, and finally summarize conclusions in Section VIII.

## II. Background

### A. Lattice of Information

An equivalence relation $\sim \subseteq A \times A$ on a set $A$ is a binary relation that is reflexive, symmetric, and transitive. For example, the equivalence relation `parity` on the set $A = \{0, 1, 2, 3\}$ is defined as $\{(x, y) \mid x, y \in A \wedge x \ mod \ 2 = y \ mod \ 2\}$. An equivalence relation partitions its underlying domain into disjoint equivalence classes. Given an equivalence relation $P$ on a set $A$ and $a \in A$, $[a]_P$ denotes the unique equivalence class induced by $P$ that $a$ belongs to. We write $[P]$ to denote the set of all equivalence classes induced by $P$. We call $[P]$ a *partition* of $A$ and hereafter we may also refer to each element, i.e. equivalence class, of the partition $[P]$ as a *cell*. For example, `parity` partitions $A$ into cells $\{0, 2\}$ and $\{1, 3\}$.

Equivalence relations over states are commonly used to represent an agent's knowledge, by relating two states whenever the agent cannot distinguish between them. When an equivalence relation models knowledge, we also call the cells induced by it *knowledge sets*. These have a distinct intuitive interpretation when we consider functions $f$ that take in some state and return an agent's *view* of it. We will write the equivalence relation induced by the output of $f$ as $\sim_f = \{(x, y) \mid f(x) = f(y)\}$. In that case, in a state $a$, the knowledge set $[a]_{\sim_f}$ represents the agent's remaining uncertainty about the state, in the sense of all the states that the agent still considers possible, after observing the output of $f$. The agent *knows* anything that is true in all states in the knowledge set. In this paper, we use the terms knowledge and information interchangeably.

A complete lattice is a set equipped with a partial ordering (reflexive, antisymmetric, and transitive) relation, maximal and minimal elements $\top$ and $\bot$ for this relation and a join (least upper bound) for any subset of elements. The meet (greatest lower bound) of a subset can be defined as the join of the set of all lower bounds of that subset [17]. The *Lattice of Information (LoI)* [7] is a structure for representing the ordering of information with equivalence relations. Let $\mathcal{L}(A)$ be the set of all equivalence relations defined on a given domain $A$. The LoI ranks these equivalence relations based on the information they reveal about the underlying domain. Given two equivalence relations $P, Q \in \mathcal{L}(A)$, this ordering can be defined as follows:

$$P \sqsubseteq Q \rightarrow \forall a, a' \in A \ \ (a \ Q \ a' \Rightarrow a \ P \ a')$$

For any set $S \subseteq \mathcal{L}(A)$, the least upper bound of $S$ is the equivalence relation $R$ defined as:

$$\forall x, y \in A\ (x\ R\ y \leftrightarrow \forall P \in S.\ x\ P\ y).$$

Formally, $LoI(A) = \langle \mathcal{L}(A), \sqsubseteq, \bigsqcup \rangle$ denotes the LoI on domain $A$, with ordering relation $\sqsubseteq$ and join $\bigsqcup$. The top element $\top$ in the lattice is the most precise equivalence relation `id` such that $\texttt{id} = \{(x, y) \mid x, y \in A \wedge x = y\}$, and the bottom element $\bot$ is the least precise equivalence relation $\texttt{all} = \{(x, y) \mid x, y \in A\}$.

The join of any two equivalence relations $P \sqcup Q$ , being their least upper bound, is the *least* informative equivalence relation that is at least as informative as either of $P$ and $Q$ (i.e. is an *upper bound* on both), and thus represents the information that is conveyed from learning both $P$ *and* $Q$. We refer to this as the *conjunction* of the information in $P$ and $Q$.

### B. Quantale of Information

The LoI captures the conjunction of any two information sources $P$ and $Q$ as the join of their respective equivalence relations. However, it does not offer an operator that would yield a representation of their *disjunction*, that is, the information that can be obtained from having access to one of them, but not both. In fact, the disjunction can not in general be represented as a single equivalence relation, and thus an element of the LoI, at all. To address this limitation, Hunt and Sands [9] propose a generalization of the LoI called the *Quantale of Information* (QoI). A quantale is a complete lattice with an additional binary "tensor" operator $\otimes$. In the QoI, the tensor is used to represent conjunction, while the lattice join represents *dis*junction.

The core idea behind the quantale structure is to interpret the disjunction $P_1 \vee \ldots \vee P_n$ of several knowledge relations as describing all knowledge relations $R$ in which the knowledge always comes from one of the $P_i$. More concretely, in any possible state $a \in A$, the agent's knowledge $[a]_R$ should equal its knowledge in the same state in one of the disjuncts, $[a]_{P_i}$. *Which* disjunct it is may depend on the state, so the agent may have knowledge from $P_i$ in the state $a$ but knowledge from $P_j$ in some other state $a'$. Relations $R$ that satisfy this condition are called *tilings*, based on a picture of covering (since every state needs to be in some equivalence class) the space of possible states $A$ with knowledge sets drawn from any of the disjuncts. Following Hunt and Sands, we define the set of all tilings

$$\mathrm{mix}(\mathbb{P}) = \{R \in LoI(A) \mid x \in [R] \Rightarrow (\exists P \in \mathbb{P}.\, x \in [P])\},$$

where $\mathbb{P}$ is a set of equivalence relations.

We would like to think of a relation $R'$ as describing no more knowledge than a disjunction $\bigvee \mathbb{P}$ if it's bounded above by *some* $R \in \mathrm{mix}(\mathbb{P})$ in the LoI, and more generally define the quantale ordering $\mathbb{S} \sqsubseteq \mathbb{T}$ for $\mathbb{S}, \mathbb{T} \subseteq \mathcal{L}(A)$ as $\forall S \in \mathbb{S},\ \exists T \in \mathbb{T}.\ S \sqsubseteq T$. The resulting relation is not antisymmetric on general sets of relations or even mixes of general sets, reflecting the circumstance that there may be multiple mixes representing the same knowledge. As it is standard in lattice theory [18], we use the downwards closure operator $\Downarrow$ to obtain canonical representations of the order cycles of $\sqsubseteq$ and hence construct a partial order.

$$\Downarrow\mathbb{P} = \{Q \in LoI(A) \mid Q \sqsubseteq \mathbb{P}\}$$

The *tiling closure* of a set of equivalence relations $\mathbb{P}$,

$$\mathrm{tc}(\mathbb{P}) = \Downarrow\mathrm{mix}(\mathbb{P}),$$

then canonically represents the knowledge permitted by the disjunction $\bigvee \mathbb{P}$. The set $\mathrm{tc}(\mathbb{P})$ can still be interpreted as a list of possible equivalence relations, now including any equivalence relation that does not reveal more information than the disjunction.

We then take the elements of the QoI on a state set $A$ to be all tiling closures of subsets of $A$, with the ordering $\sqsubseteq$ being set inclusion. For the tensor $\mathbb{P} \otimes \mathbb{Q} = \mathrm{tc}(\{P \sqcup Q \mid P \in \mathbb{P}, Q \in \mathbb{Q}\})$, we rely on the join operator of the LoI $\sqcup$ to calculate the least upper bound of any possible pair of equivalence relations in $\mathbb{P}$ and $\mathbb{Q}$ and then canonicalise the result. Since the sets are interpreted disjunctively, the join $\bigvee_i \mathbb{P}_i$ can simply be defined as $\mathrm{tc}(\bigcup_i \mathbb{P}_i)$.

```
1  if (x <= 0) then
2      out(-1  ,u);
3      out(x mod 2 == 0, u);
4  else
5      out(1, u);
6      out(x div 2 == 0, u);
```

Program 1

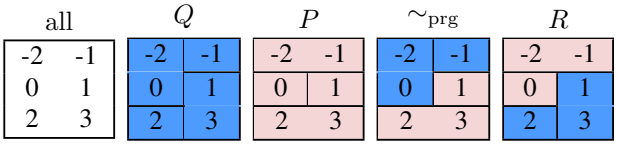


Fig. 1: Some equivalence relations on $\{-2, -1, 0, 1, 2, 3\}$

**Example 1.** Program 1 operates on a secret integer $x$ between -2 and 3, outputting to user $u$ whether it is greater than zero, and *either* (if it isn't) whether it is even, *or* (if it is) whether it equals 0 or 1 (by dividing by 2, rounding down and testing for 0). We expect the information released by the program ($\sim_{\mathrm{prg}}$ in Fig. 1) to be bounded by the disjunction of the knowledge relations capturing the two possible branches (resp. $Q$, $P$).

This could not be accurately expressed with LoI operations, since $Q$, $P$ and $\sim_{\mathrm{prg}}$ are all incomparable, but the join of $Q$ and $P$ (as the only available nontrivial way of combining them) is equal to $\top$ and so would equally bound a program that directly releases $x$. However, $\sim_{\mathrm{prg}}$ can be tiled with equivalence classes from $Q$ and $P$, and we in fact have $\mathrm{mix}(\{Q, P\}) = \{Q, P, R, \sim_{\mathrm{prg}}\}$. So in the QoI, $\mathrm{tc}(\{\sim_{\mathrm{prg}}\}) \sqsubseteq \mathrm{tc}(\{Q, P\})$, and hence $\sim_{\mathrm{prg}} \sqsubseteq Q \vee P$.

## III. Information Ordering in Databases

Our goal is to introduce our semantic model for the information revealed by database queries, the *Determinacy Lattice*, and its extension to disjunctive dependencies, the *Determinacy Quantale*. To this end, we first review a standard formalism for reasoning about databases that we will employ.

### A. A Primer on Relational Database Models

We use the relational model to formally define databases [19]. In this model, we distinguish between the database schema $D$, which specifies the structure of the database, and the database state $db$, which specifies its actual content.

A database schema $D$ is a (nonempty) finite set of relation schemas $t$, written as $D = \{t_1, ..., t_n\}$. A relation schema (table) $t$ is defined as a set of attributes paired with a set of constraints, where an attribute is a name paired with a domain. The number of attributes in $t$ (written as $|t|$) is referred to as its arity. A tuple is a set of data representing a single record within a relation schema. Each tuple contains values for each attribute defined in the relation schema.

A *database state* $db$ is a snapshot of the database schema $D$ at a particular point in time. It represents the actual data stored in the database, consisting of a collection of tables and their respective tuples. We write $[\![t]\!]^{db}$ to represent the tuples of table $t$ under database state $db$.

We write $\text{states}(D)$ to denote the set of all database states of $D$. A database configuration is $\langle D, \Gamma \rangle$ where $D$ is the database schema and $\Gamma$ is a set of integrity constraints. We denote $\Omega_D = \{db \mid db \in \text{states}(D) \wedge \vdash db : \Gamma\}$ where $\vdash$ is an appropriate notion of constraint $\Gamma$ being satisfied. An integrity constraint is an assertion about a database that must be satisfied for a database state to be considered valid. Various classes of integrity constraints exist, for instance functional dependencies which capture primary-key constraints, and inclusion dependencies which are used in foreign-key constraints [19].

**Relational calculus.** We rely on the Domain Relational Calculus (DRC) for our query language. In the DRC, a (non-boolean) query $q$ over a database schema $D$ has the form $\{\overline{x} \mid \phi\}$, where $\overline{x}$ is a sequence of variables, $\phi$ is a first order formula over $D$, and the free variables of $\phi$ are those in $\overline{x}$. The *evaluation* of a query $q$, denoted by $[\![q]\!]^{db}$, is the set of tuples that satisfy the formula $\phi$ with respect to $db$. A *boolean query* is written as $\{ \mid \phi\}$, and its evaluation $[\![q]\!]^{db}$ is defined to be the boolean value true if and only if some tuple in $db$ satisfies $\phi$. We use $\mathcal{Q}$ to indicate the universe of all possible queries.

The domain relational calculus employed here follows the standard convention, and we refer the reader to the relevant literature for a more comprehensive description of DRC [19].

**Example 2.** The database schema in Fig. 2 contains relations for employees emp and managers mng. A query returning the set of tuples containing the division names and the salary of the managers of each division can be written as:

$$\{(d, s) \mid \exists n, r.\ \text{emp}(n, r, s) \wedge \exists m.\ \text{mng}(d, m) \wedge n = m\}.$$

| | | | |
|---|---|---|---|
| emp : | name | role | salary |
| mng : | division | manager | |

Fig. 2: Database schema for employees and managers

**Views.** In DRC, a database view is a relation defined by the result of a non-boolean query. Database views act as virtual tables and, as we will see, are useful when defining security policies. Formally, a view $v$ defined over database schema $D$ is a tuple $\langle id, q \rangle$, where $id$ is the view identifier and $q$ is the non-boolean query over schema $D$ defining the view. The query $q$ may refer to other views, but we assume that views do not have cyclic dependencies.

The materialization of a view $v$ in a database state $db$ is the evaluation of its defining query $q$ in that state, i.e., $[\![q]\!]^{db}$. We use $v.q$ to refer to the defining query of view $v$. We extend relational calculus in the standard way to work with views [3].

### B. Determinacy Lattice

Given query sets $Q, Q' \in \mathcal{P}(\mathcal{Q})$, query determinacy [20] captures whether results of the queries in $Q$ are always sufficient to determine the result of the queries in $Q'$.

**Definition 1.** *$Q$ determines $Q'$ (denoted by $Q \twoheadrightarrow Q'$) iff for all database states $db_1$, $db_2$, if $[\![q]\!]^{db_1} = [\![q]\!]^{db_2}$ for all $q \in Q$, then $[\![q']\!]^{db_1} = [\![q']\!]^{db_2}$ for all $q' \in Q'$.*

Intuitively, $Q \twoheadrightarrow Q'$ means that pairs of databases for which all queries in $Q$ return the same result also give the same result under any query in $Q'$. This is in fact equivalent to the initial gloss that the results of queries in $Q'$ can be computed from the results of queries in $Q$, as we show in detail in the full version of the paper [16].

Query determinacy allows us to define an ordering on sets of queries based on the information they reveal. We call this ordering *determinacy order*, denote it by $\preceq$, and define it as $\forall Q, Q' \in \mathcal{P}(\mathcal{Q})$, $Q \preceq Q'$ iff $Q' \twoheadrightarrow Q$.

**Example 3.** Consider queries $q_1 = \{(n, r) \mid \exists s.\ \text{emp}(n, r, s)\}$ and $q_2 = \{(r) \mid \exists n, s.\ \text{emp}(n, r, s)\}$ defined on the relations of Fig. 2. Query $q_1$ discloses the name and the role of the employees while $q_2$ only returns their role. Intuitively, $q_1$ reveals more information than $q_2$, which means $q_2 \preceq q_1$.

This definition of determinacy order is a preorder (reflexive and transitive), but not necessarily a partial order, as it is not anti-symmetric. In other words, $q_1 \preceq q_2$ and $q_2 \preceq q_1$ does not necessarily mean that $q_1 = q_2$. As in Section II-A, this essentially means that query sets are not canonical representations of the information revealed by them. To rectify this, we form the closure $\downarrow$ under the determinacy order, so the determinacy order becomes set inclusion. Intuitively, $\downarrow Q$ will contain all the queries in $\mathcal{Q}$ whose answers can be inferred by the set of queries $Q$. Formally, $\downarrow Q$ is defined as:

$$\downarrow Q = \{q \in \mathcal{Q} \mid \{q\} \preceq Q\}$$

Using the definitions of determinacy order and closure $\downarrow$, we can then define the Determinacy Lattice as follows:

**Definition 2.** *Given a universe of queries $\mathcal{Q}$, the Determinacy Lattice $DL(\mathcal{Q})$ is a complete lattice $\langle \mathcal{L}, \sqsubseteq, \bigsqcup, \bot, \top \rangle$ such that:*

- $\mathcal{L} = \{\downarrow Q \mid Q \subseteq \mathcal{Q}\}$
- $\downarrow Q_1 \sqsubseteq \downarrow Q_2$ *iff* $Q_1 \preceq Q_2$
- $\bigsqcup_i \downarrow Q_i = \downarrow \bigcup_i Q_i$
- $\bot = \downarrow \varnothing$, $\top = \downarrow \mathcal{Q}$,

*where $\preceq$ is the determinacy order on $\mathcal{Q}$.*

**Disclosure order and information flow properties.** Our definition of the Determinacy Lattice is similar to the definition of the Disclosure Lattice introduced by Bender et al. [8]. A Disclosure Lattice is a lattice built upon a disclosure order, which is a partial order on sets of queries satisfying additional conditions that are expected of an ordering according to the amount of information disclosed by each set of queries. Bender et al. [8] define the disclosure order as follows:

**Definition 3.** *Given a universe of queries $\mathcal{Q}$, a disclosure order $\preceq$ is a preorder on $\mathcal{P}(\mathcal{Q})$ that satisfies the following properties:*

1) *For all $Q_1, Q_2 \in \mathcal{P}(\mathcal{Q})$, if $Q_1 \subseteq Q_2$ then $Q_1 \preceq Q_2$*
2) *If $\mathbb{P} \subseteq \mathcal{P}(\mathcal{Q})$ and $\forall P \in \mathbb{P},\ P \preceq Q$ then $\bigcup \mathbb{P} \preceq Q$*

The first property in this definition ensures that adding new elements to a set of queries only increases the amount of disclosed information and the second property allows us to derive a meaningful upper bound on the information disclosure.

The intended use of disclosure order was to order sets of queries based on the amount of information they reveal about the underlying database. However, we make the observation that this definition is not specific enough to characterize information disclosure in the information flow sense. For example, consider query containment [19], defined as:

**Definition 4.** *Given queries $q_1, q_2 \in \mathcal{Q}$, we say that $q_1$ is contained in $q_2$, denoted by $q_1 \subseteq q_2$, if for every database states $db \in \Omega_D$, we have $[\![q_1]\!]^{db} \subseteq [\![q_2]\!]^{db}$.*

Query containment satisfies all of the requirements of a disclosure order (Def. 3), but it is not enough to guarantee security. To illustrate this, consider a database with a single table $t$ given in Fig. 3.

| $vl$ |
|---|
| 0 |
| 1 |
| $100 + s$ |

Fig. 3: Table $t$

Table $t$ has a single column $vl$, and contains values 0, 1, and $100 + s$, where $s$ is a secret value that can be either 0 or 1. We thus consider two possible instances of this database, one where $t$ contains values 0, 1, and 100 and another where it contains 0, 1, and 101. Now, consider the following queries:

$$q_1 : \{(vl_1) \mid \exists vl_2.\ t1(vl_1) \wedge t_2(vl_2) \wedge vl_1 < 100\}$$
$$q_2 : \{(vl_1) \mid \exists vl_2.\ t1(vl_1) \wedge t_2(vl_2) \wedge vl_1 < 100 \wedge vl_1 = vl_2 - 100\}$$

where $t_1$ and $t_2$ are just logical copies of table $t$. It is common practice to make logical copies of relation and use them in queries with self-joins [21]. The result of query $q_1$ is always 0 and 1. The result of query $q_1$ is 1 if the secret $s$ is 1 and 0 if $s$ is 0. As it is evident, for these queries, query containment holds and the result of query $q_2$ is contained in the results of $q_1$. However, an observer seeing the result of query $q_2$ can learn the value of secret $s$.

This example illustrates that query containment (a disclosure order) is not sufficient to guarantee the confidentiality of the secret $s$ in an information flow setting. To ensure information flow security, we require a stronger condition, such as the notion of query determinacy order (Def. 1) that we chose to rely on in this paper.

**Relation between the DL and the LoI.** There exists a close relationship between the DL and the LoI. Specifically, a query $q$ defined over a database schema $D$ induces an equivalence relation $q_\sim$ on database states $db$. We can formally define this equivalence relation as:

$$q_\sim = \{(db_1, db_2) \mid db_1, db_2 \in \Omega_D \wedge [\![q]\!]^{db_1} = [\![q]\!]^{db_2}\}$$

We write $[q_\sim]$ to denote the set of all equivalence classes induced by $q$. Given an equivalence relation $q_\sim$ on set $\Omega_D$ and $db \in \Omega_D$, $[db]_{q_\sim}$ denotes the equivalence class induced by $q_\sim$ to which the database state $db$ belongs. We further lift this definition to sets of queries $Q = \{q_1, q_2, ..., q_n\}$:

$$Q_\sim = \{(db_1, db_2) \mid db_1, db_2 \in \Omega_D \bigwedge_{1 \le i \le n} [\![q_i]\!]^{db_1} = [\![q_i]\!]^{db_2}\}$$

This interpretation of database queries as equivalence relations provides a direct connection between the DL and the LoI, where the lattice elements correspond to $Q_\sim$, the ordering $\sqsubseteq$ to the determinacy order $\preceq$, and join and meet follow the definitions of the DL.

**Lemma 1.** *For all $\mathcal{Q}$, there is a complete lattice homomorphism from the Determinacy Lattice $DL(\mathcal{Q})$ to the Lattice of Information defined on $\{Q_\sim \mid Q \in DL(\mathcal{Q})\}$.*

To the extent that we believe $Q_\sim$ to accurately represent the information conveyed by the queries in $Q$, this lemma implies that joins and order comparisons can be performed in the DL without explicit reference to the LoI.

### C. Determinacy Quantale

We introduce a generalization of the Determinacy Lattice, called the *Determinacy Quantale* (DQ), to represent disjunctive dependencies. Our definition of the DQ is intended as a counterpart to the QoI [9], analogously to how the DL corresponds to the LoI. To achieve this, we define a query-set

counterpart of the tiling closure operator to capture the disjunction of sets of queries. Since *sets* of queries correspond to LoI elements (equivalence relations), disjunctive QoI elements (sets of equivalence relations) will be represented as *sets of sets* of queries. Each set of queries in the outer set represents a possible combination of queries that does not reveal more information than is allowed by the disjunction.

Analogously to the QoI, the tiling closure of a set of sets of queries is defined by forming the downward closure under $\sqsubseteq$ (from the DL) of their *mix*. The query-set equivalent of the *mix* operator is defined on a set of sets of queries $\mathbb{Q} = \{Q_1, ..., Q_n\}$ such that $Q_i \in DL(\mathcal{Q})$ for $i = 1, ...n$ as follows:

$$\text{mix}(\mathbb{Q}) = \{P \in DL(\mathcal{Q}) \mid x \in [P_\sim] \Rightarrow (\exists Q \in \mathbb{Q}. x \in [Q_\sim])\}$$

where $[Q_\sim]$ denotes the equivalence classes of $Q$ as defined previously. We then define the tiling closure for a set $\mathbb{Q}$ of elements of the DL as $\text{tc}(\mathbb{Q}) = \Downarrow\text{mix}(\mathbb{Q})$.

We then formally define the Determinacy Quantale $DQ(\mathcal{Q})$ as follows.

**Definition 5.** *Given a universe of queries $\mathcal{Q}$, let $DL(\mathcal{Q})$ be the Determinacy Lattice defined on $\mathcal{Q}$. The Determinacy Quantale $DQ(\mathcal{Q})$ is the quantale $\langle \mathcal{I}, \sqsubseteq, \bigvee, \otimes, 1\rangle$, with:*

- $\mathcal{I} = \{\text{tc}(\mathbb{Q}) \mid \mathbb{Q} \subseteq DL(\mathcal{Q})\}$
- $\bigvee_i \mathbb{P}_i = \text{tc}(\bigcup_i \mathbb{P}_i)$
- $\mathbb{P} \otimes \mathbb{Q} = \text{tc}\Big(\bigcup_{P\in\mathbb{P}, Q\in\mathbb{Q}} (P \sqcup Q)\Big)$
- $\sqsubseteq = \subseteq$
- $\top = DL(\mathcal{Q})$, $\bot = \varnothing$, $1 = \varnothing$,

*where $\mathbb{P}, \mathbb{Q} \subseteq DL(\mathcal{Q})$.*

In the full version of the paper [16], we show that Def. 5 satisfies the usual quantale axioms [9]. As with the DL and LoI, the DQ embeds into a QoI by a quantale homomorphism. This QoI is defined on sets of equivalence relations derived from sets of sets of queries by the following map:

**Definition 6.** *Given a set of sets of queries $\mathbb{Q}$,*

$$[\![\mathbb{Q}]\!] = \{Q_\sim \mid Q \in \mathbb{Q}\}.$$

We can then formally state the relationship between the DQ and this quantale as follows.

**Lemma 2.** *For all $\mathcal{Q}$, there is a quantale homomorphism from the Determinacy Quantale $DQ(\mathcal{Q})$ to the Quantale of Information defined on $\{[\![\mathbb{Q}]\!] \mid \mathbb{Q} \subseteq DL(\mathcal{Q})\}$.*

**Example 4.** To illustrate the Determinacy Quantale in practice, consider Program 2, which issues either query $q1 = \{(r, vl) \mid \exists s, n. \text{emp}(n, r, s) \wedge r = \text{Intern} \wedge vl = s\}$ or $q2 = \{(r, vl) \mid \exists s, n. \text{emp}(n, r, s) \wedge r = \text{CEO} \wedge vl = n)\}$ to the database. Query q1 returns the role and salary columns of the entry in table emp if the role of that entry is Intern. Similarly, query q2 returns the role and name columns if the role of the entry in emp is CEO.

Consider a policy defined on queries $v1 = \{(r, n) \mid \exists s. \text{emp}(n, r, s)\}$ and $v2 = \{(r, s) \mid \exists n. \text{emp}(n, r, s)\}$. v1 and v2, which respectively project on the name and role, and the role and salary columns of emp, are used in defining the disjunctive security policy $v1 \vee v2$.

```
1  if (y > 0) then
2      x ← q1
3  else
4      x ← q2
5  out(x, u);
```

Program 2

For this example, we assume a database that has only one row in the emp table, and we also limit the domain of possible roles to $\{\text{CEO}, \text{Intern}\}$. These limitations are necessary in order to have a finite representation of the potential query sets and enables us to effectively depict the sets produced by the mix and tc operators.

Program 2 depicts a disjunction that – ignoring variable y – depends either on $q1$ or $q2$ (i.e., $q1 \vee q2$), which on the DQ can be represented as a point $\text{tc}(\downarrow\{q1\}) \vee \text{tc}(\downarrow\{q2\})$. Similarly, the policy $v1 \vee v2$ can be represented on the DQ by $\text{tc}(\downarrow\{v1\}) \vee \text{tc}(\downarrow\{v2\})$.

Illustrating this point requires calculating the mix set of v1 and v2, which includes all sets of queries whose equivalence relation can be constructed from the equivalence classes of $\downarrow\{v1\}_\sim$ and $\downarrow\{v2\}_\sim$. Unfortunately, for any sufficiently rich query language, our definition of mix inevitably yields an infinite set, as infinitely many queries that are "morally equivalent" or even the same up to renaming variables represent the same knowledge set. To compactly represent such infinite sets, we will pick just one representative, and define

$$hc(\mathbb{Q}) = \{Q' \mid \exists Q \in \mathbb{Q}.\ Q_\sim = Q'_\sim\}$$

as a closure operator that adds all equivalent queries. Then $\text{mix}(\{\downarrow\{v1\}, \downarrow\{v2\}\})$ will be the set $hc(\{\downarrow\{v1\}, \downarrow\{v2\}, \downarrow\{p1\}, \downarrow\{p2\}\})$, where $p1 = \{(r, vl) \mid \big(\exists s, n. \text{emp}(n, r, s) \wedge r = \text{Intern} \wedge vl = s\big) \vee \big(\exists s, n. \text{emp}(n, r, s) \wedge r = \text{CEO} \wedge vl = n\big)\}$ and $p2 = \{(r, vl) \mid \big(\exists s, n. \text{emp}(n, r, s) \wedge r = \text{CEO} \wedge vl = s\big) \vee \big(\exists s, n. \text{emp}(n, r, s) \wedge r = \text{Intern} \wedge vl = n\big)\}$.

Therefore, we can depict the policy as the point $\Downarrow(hc(\{\downarrow\{v1\}, \downarrow\{v2\}, \downarrow\{p1\}, \downarrow\{p2\}\}))$ on the DQ. Similarly, the DQ point of the Program 2 (i.e., $\text{tc}(\downarrow\{q1\}) \vee \text{tc}(\downarrow\{q2\})$), can also be depicted by the point $\Downarrow hc(\{\downarrow\{p1\}\})$ on the DQ. We illustrate the part of the DQ which includes these points in Fig. 4, and as it is evident from the figure, conclude that Program 2 is inline with the policy.

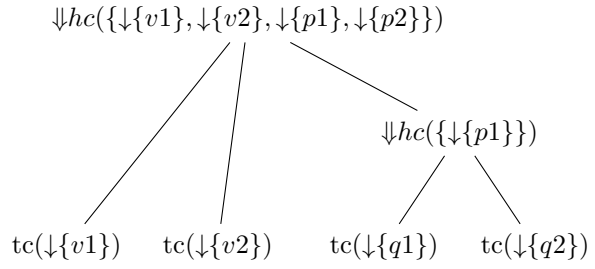


Fig. 4: A portion of the DQ for queries q1, q2, v1, v2

$$
\begin{aligned}
c := \quad & \texttt{skip} \mid \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 \mid \\
& x \leftarrow q \mid x := e \mid c_1; c_2 \mid \\
& \texttt{while } e \texttt{ do } c \mid \texttt{out}(e, u)
\end{aligned}
$$

Fig. 5: Language syntax

## IV. Security Framework

Drawing on the quantale model of dependencies for programs and databases, we develop an extensional condition that defines security for programs that interact with databases and support disjunctive security policies. We will later use the security condition to prove soundness of enforcement mechanisms in Section V. Specifically, we formalize the syntax and semantics of a simple imperative language with database queries. Programs read the input from the database via queries, while users receive the output through predefined output channels. We define (disjunctive) security policies as views over the database and interpret them end-to-end. We then use this model to define a knowledge-based security condition for our setting.

### A. Language

**Syntax.** The syntax for the commands of our language as depicted in Fig. 5, primarily consists of standard commands such as assignment, conditionals, and loops. The command $\texttt{out}(e, u)$ outputs the result of evaluating expression $e$ to user $u \in \mathcal{U}$. The command $x \leftarrow q$ issues the query $q$ to the database and stores the result in variable $x$. For modeling the queries, we rely on conjunctive queries with comparison introduced in Section V-A.

Expressions $e$ can be variables $x \in \text{Vars}$, values (integers) $n \in \text{Val}$, binary operations $e_1 \oplus e_2$, single tuples $tp \in \text{Val}$, and set of tuples $\overline{tp} \in \text{Val}$. For simplicity, we do not provide de-constructors for database tuples.

**Semantics.** As discussed in Section III-C, a database state (or simply state) $db \in \Omega_D$ is defined with respect to a schema $D$ and a finite set of integrity constraints. A configuration $\langle c, m, db\rangle$ consists of a command $c$, a memory $m = \text{Var} \to \text{Val}$ mapping variables to values, and a state $db$.

The semantics of expressions is mostly standard and its rules are presented in Fig. 6. We use judgments of the form $\langle e, m, db\rangle \downarrow vl$ to denote that an expression $e$ evaluates to value $vl$ in memory $m$ and state $db$. For simplicity, we refrain from defining binary operations on tuples, unless the underlying database query is boolean.

We use judgments of the form $\langle c, m, db\rangle \xrightarrow{\alpha} \langle c', m', db'\rangle$ to denote that a configuration $\langle c, m, db\rangle$ in one step evaluates to memory $m'$ and state $db'$ and (possibly) produces an observation $\alpha \in \text{Obs}$; we write $\epsilon$ whenever a command produces no observation. We write $m[x \mapsto vl]$ to denote a memory $m$ with variable $x$ assigned the value $vl$.

Fig. 7 provides the semantic rules for commands. The query evaluation rule QueryEval is similar to assignment as it evaluates a query $q$ into state $db$ and stores the result in the variable $x$. We use the command $\texttt{out}(e, u)$ to produce an observation. Formally, an observation $\alpha \in \text{Obs}$ is a tuple $\langle o, u\rangle$, where $u \in \mathcal{U}$ is the identifier of the user observing the output and $o$ is the result of evaluating expression $e$, which is either a simple value or the result set of a non-boolean query.

We write $\langle c, m, db\rangle \overset{\tau}{\Rightarrow}_u \langle c', m', db'\rangle$ to denote when $\langle c, m, db\rangle$ takes one or more steps to reach configuration $\langle c', m', db'\rangle$ while producing the trace (sequence of observations) $\tau \in \text{Obs}^*$. We omit the final configuration whenever it is irrelevant and write $\langle c, m, db\rangle \overset{\tau}{\Rightarrow}_u$.

### B. Security Model

We now introduce our knowledge-based security model for disjunctive security policies. For simplicity, we denote the initial program memory by $m_0$ and assume it is fixed and public to all users, hence the only way to input sensitive information is through database queries. Users make observations through output channels, hence their knowledge of the database is determined by what they can infer based on these observations. This model induces standard equivalence relations for database states and observation traces.

**Database state equivalence.** Two states $db$ and $db'$ are equivalent with respect to a set of tables and views $V$, written as $db \approx_V db'$, iff all tables and views in $V$ have identical contents in $db$ and $db'$. Formally, states $db$ and $db'$ are equivalent with respect to $V$ iff for all view $v \in V$, $[\![v.q]\!]^{db} = [\![v.q]\!]^{db'}$ and for all table $t \in V$, $[\![t]\!]^{db} = [\![t]\!]^{db'}$. A set of tables and views $V$ induces an equivalence relation, and for a state $db$, the equivalence class $[db]_V$ contains all states that are equivalent to $db$ with respect to $V$.

**Trace equivalence.** We use trace projection to define trace equivalence. The projection of a trace $\tau$ for user $u$ written as $\tau\lfloor_u$ is the sequence of all observations in $\tau$ that $u$ can observe. Traces $\tau_1$ and $\tau_2$ are equivalent with respect to user $u$, written as $\tau_1 \approx_u \tau_2$, iff the projection of one of them to $u$ is the prefix of the other, i.e., $\tau_1\lfloor_u \preceq \tau_2\lfloor_u$ or $\tau_1\lfloor_u \succeq \tau_2\lfloor_u$.

Equivalence of trace prefixes is a standard technicality needed to ignore leaks due to program's progress/termination [22], and here we adapt a definition of trace equivalence which does not differentiate between program divergence and termination [14].

**User knowledge.** When executing a program prg, we assume memory is always initially in the all-zero state $m_0$. Thus, we can view a program's execution for any user as a function from database $db$ to user-observable output traces, $\tau_{\text{prg},u}(db) = \tau\lfloor_u$ when $\langle \text{prg}, m_0, db\rangle \overset{\tau}{\Rightarrow}_u$. This function induces an equivalence relation on databases, $[\![\text{prg}]\!]_u = \sim_{\tau_{\text{prg},u}}$, which characterizes the knowledge of $db$ conveyed by the output of prg to $u$.

**Security policy.** A security policy is a list of user policies (written as $P_u$) for each user $u \in \mathcal{U}$. User policies are defined as views and table identifiers over a database schema, and determine what a user $u$ is allowed to observe. Fig. 8 presents the syntax of disjunctive policies for our model. They are

$$\textsc{Int}\ \frac{}{\langle n, m, db\rangle \downarrow n} \qquad \textsc{Tuple}\ \frac{}{\langle tp, m, db\rangle \downarrow tp} \qquad \textsc{TupleSet}\ \frac{}{\langle \overline{tp}, m, db\rangle \downarrow \overline{tp}} \qquad \textsc{Var}\ \frac{vl = m(x)}{\langle x, m, db\rangle \downarrow vl}$$

$$\textsc{Op}\ \frac{\langle e_1, m, db\rangle \downarrow n_1 \qquad \langle e_1, m, db\rangle \downarrow n_2 \qquad n = n_1 \oplus n_2}{\langle e_1 \oplus e_2, m, db\rangle \downarrow n}$$

Fig. 6: Semantic rules for expressions

$$\textsc{Skip}\ \frac{}{\langle \texttt{skip}, m, db\rangle \xrightarrow{\epsilon} \langle \epsilon, m, db\rangle} \qquad \textsc{Assign}\ \frac{\langle e, m, db\rangle \downarrow vl \qquad m' = m[x \mapsto vl]}{\langle x := e, m, db\rangle \xrightarrow{\epsilon} \langle \epsilon, m', db\rangle} \qquad \textsc{QueryEval}\ \frac{vl = [\![q]\!]^{db} \qquad m' = m[x \mapsto vl]}{\langle x \leftarrow q, m, db\rangle \xrightarrow{\epsilon} \langle \epsilon, m', db\rangle}$$

$$\textsc{IfTrue}\ \frac{\langle e, m, db\rangle \downarrow n \qquad n \neq 0}{\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, m, db\rangle \xrightarrow{\epsilon} \langle c_1, m, db\rangle} \qquad \textsc{IfFalse}\ \frac{\langle e, m, db\rangle \downarrow n \qquad n = 0}{\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, m, db\rangle \xrightarrow{\epsilon} \langle c_2, m, db\rangle}$$

$$\textsc{WhileTrue}\ \frac{\langle e, m, db\rangle \downarrow n \qquad n \neq 0}{\langle \texttt{while } e \texttt{ do } c, m, db\rangle \xrightarrow{\epsilon} \langle c; \texttt{while } e \texttt{ do } c, m, db\rangle} \qquad \textsc{WhileFalse}\ \frac{\langle e, m, db\rangle \downarrow n \qquad n = 0}{\langle \texttt{while } e \texttt{ do } c, m, db\rangle \xrightarrow{\epsilon} \langle \epsilon, m, db\rangle}$$

$$\textsc{Seq}\ \frac{\langle c_1, m, db\rangle \xrightarrow{\alpha} \langle c_1', m', db'\rangle}{\langle c_1; c_2, m, db\rangle \xrightarrow{\alpha} \langle c_1'; c_2, m', db'\rangle} \qquad \textsc{SeqEmpty}\ \frac{}{\langle \epsilon; c, m, db\rangle \xrightarrow{\epsilon} \langle c, m, db\rangle} \qquad \textsc{Output}\ \frac{\langle e, m, db\rangle \downarrow vl}{\langle \texttt{out}(e, u), m, db\rangle \xrightarrow{\langle vl, u\rangle} \langle \epsilon, m, db\rangle}$$

Fig. 7: Semantics rules for commands

$$\begin{aligned} \text{con} &:= \{v\} \mid \{t\} \mid \text{con}_1 \cup \text{con}_2 \\ \text{dis} &:= \{\text{con}\} \mid \text{dis}_1 \cup \text{dis}_2 \\ P_u &:= \text{dis} \end{aligned}$$

Fig. 8: Syntax of user policy

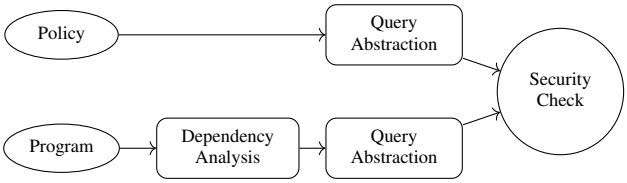


Fig. 9: Enforcement steps

defined as a set of sets in order to represent a disjunction of conjunctions of simpler policies. A conjunction con is a set of view $v$ and table $t$ identifiers, and a disjunction dis is a set of conjunctions. For example, the policy $P_u$ for user $u$ who is allowed to see table $t_1$ and view $v_1$, or view $v_2$ but not both, is defined as $P_u = \{\{t_1, v_1\}, \{v_2\}\}$.

The overall policy of the system, written as $P$, is the list of user policies. Per Def. 6, the policy $P_u$ can be represented semantically as an element $[\![P_u]\!]$ of the Quantale of Information. Thus, we can formulate our security condition as the assertion that the knowledge of the database that the execution of the program prg conveys to $u$ is bounded above by the disjunctive knowledge allowed by the policy, $[\![P_u]\!]$.

**Definition 7.** *The program* prg *is secure for the user* $u$ *and policy* $P_u$ *if* $[\![\text{prg}]\!]_u \sqsubseteq [\![P_u]\!]$.

## V. Enforcement of Disjunctive Policies

Having formulated the security condition, we would like to prove that useful programs satisfy it. To this end, we introduce a sound static enforcement mechanism, which imposes some structural limitations on the policy and trades off some completeness for the sake of efficiency and ease of analysis.

Fig. 9 illustrates how our mechanism functions at a high level. We assume as input a program and policy in the format described in Fig. 5 and Fig. 8 respectively. The program is then subjected to a static *dependency analysis* (Section V-B), which computes an overapproximate set of possible paths of control flow through the program, along with the queries (dependencies) retrieved for each path, giving an element of the DQ, that is a (disjunctive) set of (conjunctive) sets of queries. Per Fig. 8, the policy is also already given in this format.

We would like to verify that the program dependencies are bounded by the policy in the DQ, as by Lemma 2, this entails the security condition (Def. 7) that the disjunctive information that is revealed by the program is bounded above by the QoI interpretation of the policy. However, checking DQ ordering on general queries may be computationally costly. We therefore *abstract* (Section V-C) both the policy and the path dependencies into a more tractable format (symbolic tuples), which again overapproximates the information they can retrieve. To guarantee soundness, we require that the views in the policy are such that this abstraction is lossless for them. Finally, as the *security check* (Section V-D), we compute a tractable comparison on sets of sets of symbolic tuples that can be shown to imply DQ ordering.

### A. Conjunctive Queries

While our theoretical definitions are based on the fully-general domain relational calculus as a query language, to avoid complexity, our enforcement mechanism will work with a restricted subset called *conjunctive queries with comparisons* (CQCs). This language is a subset of relational calculus that only employs conjunction ($\wedge$) and existential quantification ($\exists$) and omits disjunction ($\vee$), negation ($\neg$), and universal quantification ($\forall$). CQCs can model `SELECT-FROM-WHERE` portion of SQL, where there are only `AND` and comparisons in the `WHERE` clause.

Our language for (non-boolean) CQC $q$ over a database schema $D$ employs the standard notation [19], [21], and has the form *heading* $\leftarrow$ *body*:

$$\text{ans}(\overline{y}) \leftarrow R_1(\overline{x}_1), ..., R_n(\overline{x}_n), C_1, ..., C_m$$

where $R_1, ..., R_n$ are relations in $D$, and $\overline{x}_1, ..., \overline{x}_n$ are their variables. We use $\text{Var}(q) = \overline{x}_1 \cup ... \cup \overline{x}_n$ to denote the set of variables appearing in the body of the query $q$. $C_1, ..., C_m$ are formulae of the form $x_i \oplus x_j$ where $\oplus$ is the comparison operator which could be anything from $<, \leq, =, \neq, >, \geq$ and $x_i$ and $x_j$ are either variables in $\text{Var}(q)$ or constants.

We require that $\overline{y} \subseteq \text{Var}(q)$. Without loss of generality, we assume that there are no self-joins in the query. In case of queries with self-joins, we can make logical copies of the relations to accommodate them [21]. The body of a CQC $q$ comprises two parts, namely the relation identifiers $R_1, ..., R_n$ referred to as $\text{ids}(q)$, and the conditions $C_1, ..., C_m$ denoted by $\text{cnd}(q)$.

Similarly to Section III-A, the evaluation of $q$ on the database state $db$ (denoted by $[\![q]\!]^{db}$) is defined by taking all tuples in the cartesian product of $\text{ids}(q)$ in $db$ that satisfy $\text{cnd}(q)$, and projecting to the column set $\overline{y}$.

**Example 5.** Consider the database schema in Fig. 2. The following query returns a set of tuples containing the names of divisions whose managers have a salary of more than 50:

$$\text{ans}(d) \leftarrow \text{emp}(n, r, s), \text{mng}(d, m), n = m, s > 50$$

### B. Type-based Dependency Analysis

Our static dependency analysis builds on the generic type system of van Delft et al. [15] and extends it with support for disjunctive dependencies. We intuitively expect that a disjunctive dependency analysis must be path-sensitive, so as to distinguish between different executions and also keep track of the history of observations. Both of these requirements are often challenging for type-based analyses, which do not naturally align with the execution order. We will first illustrate these challenges with examples and then present our analysis.

Program 3 illustrates the need for path sensitivity. The analysis should distinguish between the *then* branch, where variable $x$ depends on the set $\{y, w, z\}$, and the *else* branch where $x$ depends on $\{y, x\}$. Our reference analysis [15] would join these two sets at the end of the if statement, ultimately yielding the dependency set $\{x, y, w, z\}$. In our analysis,

```
1  if (y > 0) then
2      x := w + z;
3  else
4      x := x + 1;
5  out(x,u);
```

Program 3

```
1  if (z == 0) then
2      x ← q1;
3  else
4      x ← q2;
5  out(x,u);
6  if (z != 0) then
7      x ← q1;
8  else
9      x ← q2;
10 out(x,u);
```

Program 4

these sets are never joined, but instead combined to form a set of sets, namely, $\{\{y, w, z\}, \{y, x\}\}$, where the outer set represents a disjunctive dependency and the inner sets represent conjunctive dependency.

Program 4 illustrates the need to keep track of the observation history. It outputs $x$ at lines 5 and 10, and the dependency set of $x$ in both places is $\{\{q1, z\}, \{q2, z\}\}$. However, this program will always output both $q1$ and $q2$. Now, if a policy only allows user $u$ to see either query $q1$ or $q2$, the outputs at lines 5 and 10 will be incorrectly accepted. Hence, the analysis should account for all outputs to user $u$.

Fig. 10 depicts the rules of our disjunctive dependency analysis. We use judgments of the form $\vdash c : \Gamma$, where $\Gamma$ is an environment mapping variables Var to set of sets of dependencies Dep. The set of variables is $\text{Var} = PV \cup \mathcal{U} \cup \{pc\}$, where $PV$ are program variables, $\mathcal{U}$ are users, and $pc$ is the program context. The dependencies Dep are $\text{Dep} = \text{Var} \cup \mathcal{Q}$, where Var are variables and $\mathcal{Q}$ are queries that can be issued to a database. We use $u \in \mathcal{U}$ to indicate the dependencies of all outputs to user $u$.

We start by introducing the operators and auxiliary functions employed within the rules, and then proceed to explain the rules themselves. The operator $\otimes$ is used to join two (or more) sets of sets, defined as:

$$\Gamma_1(x_1) \otimes ... \otimes \Gamma_n(x_n) = \{S_1 \cup ... \cup S_n \mid S_i \in \Gamma_i(x_i) \quad i = 1, \ldots, n\}$$

For example, the join of $\Gamma_1(x) = \{\{x, y\}, \{z, y\}\}$ and $\Gamma_2(y) = \{\{w\}, \{x, z\}\}$ is:

$$\Gamma_1(x) \otimes \Gamma_2(y) = \{\{x, y, w\}, \{x, y, z\}, \{z, y, w\}\}$$

Intuitively, the result of the join operator is a set of sets capturing the product of the original sets of sets under the set union operation. We use this operator to calculate all the possible combinations of two environments.

$\Gamma_2; \Gamma_1$ represents the sequential composition of two environments. Intuitively, $\Gamma_2; \Gamma_1$ is the same as $\Gamma_2$ but updated with all of the dependencies that have been previously established in $\Gamma_1$. Formally:

$$\Gamma_2; \Gamma_1(x) = \bigcup_{S_2 \in \Gamma_2(x)} \bigotimes_{y \in S_2} \Gamma_1(y)$$

For example, the sequential composition of the environments

$$\Gamma_1 = [x \mapsto \{\{x\},\{y\}\}, y \mapsto \{\{y\}\}, pc \mapsto \{\{y, pc\}\}]$$
$$\Gamma_2 = [x \mapsto \{\{pc, x\}\}, y \mapsto \{\{pc, y\}\}, pc \mapsto \{\{pc\}\}]$$

evaluates to

$$\Gamma_2;\Gamma_1 = [x \mapsto \{\{x, y, pc\},\{y, pc\}\}, y \mapsto \{\{pc, y\}\}, pc \mapsto \{\{y, pc\}\}]$$

Finally, the operator $\Cup$ calculates the union of two environments: $\Gamma_1 \Cup \Gamma_2 = \forall x \in \text{Var},\ \Gamma_1(x) \cup \Gamma_2(x)$. This operator is used in conditionals to capture the disjunctive join of the two branches. For example, in line 5 in Program 3, $\Gamma_1(x) = \{\{y, w, z\}\}$ and $\Gamma_2(x) = \{\{y, x\}\}$, and the result of $(\Gamma_1 \Cup \Gamma_2)(x)$ would be $\{\{y, w, z\}, \{y, x\}\}$.

For loops, we rely on the fixed-point of $\Gamma$, denoted by $\Gamma^*$, which we define as:

$$\Gamma^* = \bigcup_{n>0} \Gamma^n$$

where $\Gamma^0 = \Gamma_{id}$ and $\Gamma^{n+1} = \Gamma^n;\Gamma$.

In these rules, $\Gamma_{id}$ is the identity environment, defined as $\forall x \in \text{Var},\ \Gamma_{id}(x) = \{\{x\}\}$, and $fv(e)$ denotes the free variables of expression $e$.

T-ASSIGN updates the dependency set of the assigned variable $x$ to the set of the free variables of expression $e$ and $pc$, otherwise it matches the identity environment. Rule T-QUERYEVAL is similar to assignment, except that instead of $fv(e)$, it adds query $q$ to the dependency set.

T-IF sequentially composes the dependency sets of each branch with the environment $\Gamma_{id}[pc \mapsto \{fv(e) \cup \{pc\}\}]$, thus adding variables of the branch condition to the dependency set of each branch. Finally, these environments ($\Gamma_1$ and $\Gamma_2$) are joined disjunctively using the $\Cup$ operator.

T-WHILE uses the fixed-point operator to calculate the dependency set of the loop. To do so, it first calculates the dependency set of the loop body, which is sequentially composed with $\Gamma_{id}[pc \mapsto \{fv(e) \cup \{pc\}\}]$ to account for the dependencies to the loop condition. Finally, the fixed-point operator computes the dependency set of the while loop.

T-OUTPUT relies on the dependency set including $fv(e)$, $\{pc\}$ and $\{u\}$, where $fv(e)$ includes all the variables of the expression outputted to user $u$, $\{pc\}$ captures the implicit dependencies to the path conditions, and $\{u\}$ is the dependency set of user $u$ and captures the history of dependencies that user $u$ might have observed up to this point. Observe that by the definition of sequential composition, all the dependencies of the previous outputs will be added to $u$.

This analysis yields a final environment $\Gamma_{\text{fin}}$. The result of the analysis is the value of this environment for the user identifier $u$, which includes both queries and program variables. Since program variables do not contain sensitive information, and we are primarily concerned with queries, we refine the result of $\Gamma_{\text{fin}}(u)$ to only include queries. This refined outcome defines the ultimate result of our analysis, denoted as $\text{QL}_u$:

$$\text{QL}_u \triangleq \bigcup_{S \in \Gamma_{\text{fin}}(u)} \{S \cap \mathcal{Q}\}$$

The soundness proof of our enforcement relies on the circumstance that, if the set of queries on which the $u$-outputs of prg depend when running on a database state $db$ are denoted by $Q_{\text{prg},u}(db)$, then this set is guaranteed to be found in the set $\text{QL}_u$ produced by the dependency analysis. We show how to define $Q_{\text{prg},u}(db)$ using a taint-tracking semantics presented in the full version of the paper [16]. Formally, this gives rise to the following soundness condition for the dependency analysis.

**Lemma 3.** *For all $db \in \Omega_D$, $Q_{\text{prg},u}(db) \in \text{QL}_u(\text{prg})$.*

### C. Query Abstraction

Even for CQCs, comparing the information revealed by sets of queries is hard in general. To define a well-behaved and more tractable determinacy order on which to build our DQ, we introduce another overapproximating abstraction, which we will use to soundly *label* queries and policies.

We define a *symbolic tuple* as $\langle T, \phi, \pi\rangle$, where $T = \{t_1, t_2 ..., t_n\}$ is a set of table identifiers, $\phi$ is a boolean combination of equality, inequality, and comparisons over the columns of the tables in $T$, and $\pi$ is a subset of the columns of the tables in $T$. In a symbolic tuple, $\pi$ denotes the query's projection on the columns of the tables in $T$, and $\phi$ defines the constraints over the rows.

**Example 6.** The symbolic tuple of query $\text{ans}(d) \leftarrow \text{emp}(n, r, s), \text{mng}(d, m), n = m, s > 50$ defined on the relations of Fig. 2 would be $\langle\{\text{emp}, \text{mng}\}, s > 50 \wedge n = m, \{d\}\rangle$.

While calculating the exact set of symbolic tuples of a relational calculus query is intractable for many classes of queries, it is tractable for conjunctive queries with comparison (CQC). Given a conjunctive query $q = \text{ans}(\overline{y}) \leftarrow R_1(\overline{x}_1), ..., R_n(\overline{x}_n), C_1, ..., C_m$, the function sts computes a symbolic tuple from $q$ as follows:

$$\text{sts}(q) = \langle \text{ids}(q'), (\bigwedge_{C \in \text{cnd}(q')} C\ ), \overline{y}\rangle$$

where $\text{ids}(q')$ and $\text{cnd}(q')$ defined in Section V-A return the relation identifiers and conditionals of $q'$, respectively. Here, $q'$ is the query obtained by recursively replacing views with their definitions. We lift this definition to sets of queries $Q$, and define $\text{sts}(Q)$ as $\{\bigcup_{q \in Q} \text{sts}(q)\}$.

Using sts, we define the function $\sigma_{\text{st}}$ for a set of sets of queries $\mathbb{Q}$ as follows:

$$\sigma_{\text{st}}(\mathbb{Q}) = \{\text{sts}(Q) \mid Q \in \mathbb{Q}\}$$

**Policy Analysis.** The function $\sigma_{st}$ can also be used to map a disjunctive security policy to a set of labels. However, in order to ensure soundness and avoid approximation, we place some constraints on policies. (1) To make computing the set of symbolic tuples tractable we only support policies with views

$$\text{T-Skip}\ \frac{}{\vdash \texttt{skip} : \Gamma_{id}} \qquad \text{T-Assign}\ \frac{\Gamma = \Gamma_{id}[x \mapsto \{fv(e) \cup \{pc\}\}]}{\vdash x := e : \Gamma} \qquad \text{T-Output}\ \frac{\Gamma' = \Gamma_{id}[u \mapsto \{fv(e) \cup \{pc, u\}\}]}{\vdash \texttt{out}(e, u) : \Gamma'}$$

$$\text{T-QueryEval}\ \frac{\Gamma = \Gamma_{id}[x \mapsto \{\{q, pc\}\}]}{\vdash x \leftarrow q : \Gamma} \qquad \text{T-If}\ \frac{\vdash c_i : \Gamma_i \qquad \Gamma'_i = \Gamma_i; \Gamma_{id}[pc \mapsto \{fv(e) \cup \{pc\}\}]\ i = 1, 2 \qquad \Gamma' = (\Gamma'_1 \Cup \Gamma'_2)[pc \mapsto \{\{pc\}\}]}{\vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : \Gamma'}$$

$$\text{T-While}\ \frac{\vdash c : \Gamma_c \qquad \Gamma_f = (\Gamma_c; \Gamma_{id}[pc \mapsto \{fv(e) \cup \{pc\}\}])^* \qquad \Gamma' = \Gamma_f[pc \mapsto \{\{pc\}\}]}{\vdash \texttt{while}\ e\ \texttt{do}\ c : \Gamma'} \qquad \text{T-Seq}\ \frac{\vdash c_1 : \Gamma_1 \qquad \vdash c_2 : \Gamma_2 \qquad \Gamma' = \Gamma_2; \Gamma_1}{\vdash c_1; c_2 : \Gamma'}$$

Fig. 10: Type-based dependency analysis rules

in the CQC form. (2) We require that the symbolic tuples of views be *well-formed*, which we define as:

**Definition 8.** *The symbolic tuple* $\langle T, \phi, \pi \rangle$ *is said to be well-formed if it satisfies* $\text{dep}(\phi) \subseteq \pi$.

where $\phi = C_1 \wedge ... \wedge C_n$ and $\text{dep}(\phi) = \bigcup_{i \in \{1,...,n\}} fv(C_i)$ returns the column dependency set of $\phi$.

Well-formedness ensures that the symbolic tuples are precise, at the expense of limiting a view to only applying constrains on the columns which it projects on.

Furthermore, we treat the table identifiers used in policies as special views that return the whole table. For instance, a policy which allows access to table emp can be rewritten as view $\text{ans}(n, r, s) \leftarrow \text{emp}(n, r, s)$.

As discussed in Section IV, the disjunctive security policy of user $u$ (written as $P_u$) is a set of conjunctions con, interpreted as a disjunction of conjunctions of table and view identifiers. For a policy $P_u$ that adheres to the constraints mentioned earlier, $\sigma_{\text{st}}$ is defined as follows:

$$\sigma_{\text{st}}(P_u) = \{\text{sts}(\text{con}) \mid \text{con} \in P_u\}$$

**Labels.** In our model, a security label $\ell$ is defined as a set of symbolic tuples, and we define the ordering relation of two labels, written as $\ell_1 \sqsubseteq_{\text{st}} \ell_2$, as follows:

**Definition 9.** $\ell_1 \sqsubseteq_{\text{st}} \ell_2$ *iff for all symbolic tuples* $\langle T, \phi, \pi \rangle \in \ell_1$, *there are well-formed symbolic tuples* $\langle T_1, \phi_1, \pi_1 \rangle, ..., \langle T_n, \phi_n, \pi_n \rangle$ *in* $\ell_2$ *such that* $T \subseteq (T_1 \cup ... \cup T_n)$, $T_1, ..., T_n$ *are disjoint,* $\phi \models (\phi_1 \wedge ... \wedge \phi_n)$, *and* $dep(\phi) \cup \pi \subseteq (\pi_1 \cup ... \cup \pi_n)$.

To ensure soundness, we assume that all of the symbolic tuples in the right hand side of $\sqsubseteq_{\text{st}}$ are well-formed. This definition relies on entailment to check the ordering of $\phi$, and write $\phi_1 \models \phi_2$ which means that any assignment that satisfies $\phi_1$ also satisfies $\phi_2$.

**Example 7.** Consider symbolic tuples $\ell_1 = \{\langle \{\text{emp}\}, s = 10, \{r\} \rangle\}$ and $\ell_2 = \{\langle \{\text{emp}, \text{mng}\}, s > 5, \{r, s, m\} \rangle\}$. We have $\ell_1 \sqsubseteq_{\text{st}} \ell_2$ since $\{\text{emp}\} \subseteq \{\text{emp}, \text{mng}\}$, $\{r\} \subseteq \{r, s, m\}$, $s = 10 \models s > 5$ and $\{s\} \cup \{r\} \subseteq \{r, s, m\}$.

### D. Enforcement

The dependency analysis of Section V-B extracts the dependencies of program prg's outputs to user $u$ and produces $\text{QL}_u$. Applying $\sigma_{\text{st}}$ to $\text{QL}_u$ yields a set of labels, each bounding the information revealed in some path, the $u$-knowledge of prg (denoted by $k(\text{prg})_u$). We interpret this as a disjunction, as any execution follows along one particular path.

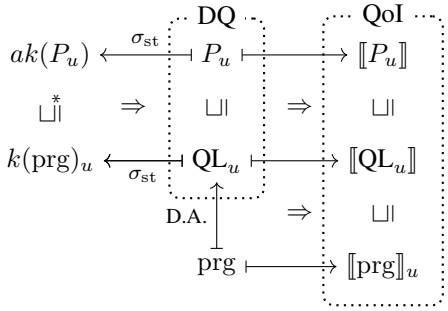


Fig. 11: Overall architecture of our proof

Similarly, applying $\sigma_{\text{st}}$ to the disjunctive security policy of user $u$ (i.e., $P_u$) results in a set of labels. Each label faithfully captures one conjunction, and so the policy is also represented as a set of labels $ak(P_u)$, interpreted disjunctively.

By Lemma 2, to verify that the security condition is satisfied, it is sufficient to establish that $\text{QL}_u \sqsubseteq P_u$ in the DQ. However, checking $\sqsubseteq$ in the DQ is not generally tractable. For the security check, we therefore instead perform a twofold approximation: we check ordering for the conjunctive inner sets using the approximate ordering $\sqsubseteq_{\text{st}}$, and approximate the mix-based ordering on the disjunctive outer sets in a way that loses little relative to our analysis:

**Definition 10.** *We say that* $k(\text{prg})_u \sqsubseteq_* ak(P_u)$ *iff*

$$\forall \ell_k \in k(\text{prg})_u,\ \exists \ell_{ak} \in ak(P_u).\ \ \ell_k \sqsubseteq_{\text{st}} \ell_{ak}$$

where $\ell_{ak}$ and $\ell_k$ are labels, and $\sqsubseteq_{\text{st}}$ is the symbolic tuple ordering of Def. 9. To ensure faithful labeling of policies, we assume all of the symbolic tuples in $\ell_{ak}$ are well-formed as defined in Def. 8. We can then formalize the relationship between $\sqsubseteq_*$ and $\sqsubseteq$ as follows.

**Lemma 4.** *If* $\sigma_{\text{st}}(\{Q_1, ..., Q_n\}) \sqsubseteq_* \sigma_{\text{st}}(\{P_1, ..., P_m\})$, *then in the DQ,* $(Q_1 \vee ... \vee Q_n) \sqsubseteq (P_1 \vee ... \vee P_m)$.

### E. Soundness Proof

Fig. 11 outlines the overall architecture of our enforcement mechanism and the correctness assertion that we make of it.

The rightmost column of Fig. 11 represents a chain of information order relations in the QoI, which we establish for

each enforcement step. Following the chain from bottom to top, we obtain the security condition of Def. 7. At the same time, the "left boundary" of the figure, comprising the D.A., $\sigma_{\text{st}}$ abstractions and $\sqsubseteq_*$ check, represents the computations that are actually performed to check a program.

**Theorem 1.** *If a program* prg *satisfies Def. 10, then it is secure in the sense of Def. 7.*

## VI. Implementation and Evaluation

In this section, we describe our prototype DiVerT [23], which implements the type-based dependency analysis of Section V-B and query abstraction of Section V-C to verify the security of database-backed programs. We then evaluate DiVerT's effectiveness using functional tests and an assortment of real-world-inspired use cases.

### A. Implementation

To evaluate the feasibility and security of our approach in practice, we implemented the type-based dependency analysis of Section V-B. For the sake of practicality, instead of CQC, DiVerT uses the `SELECT-FROM-WHERE` portion of SQL, which is analogous to CQC as described in Section V-A. Following the query analysis of Section V-C, these SQL queries are then converted into symbolic tuples. For the security check, the symbolic tuples with the result of the program analysis must be compared to those representing the policy; to perform this comparison following Def. 9, we use the Z3 SMT solver [24]. Our implementation operates on programs in the language presented in Section IV-A, with the addition of two macros `@Table@` and `@Policy@` for defining the tables' schema and the security policy.

### B. Test suite

To validate our implementation, we use a functional test suite consisting of 20 programs, designed to capture a broad variety of examples of disjunctive dependencies. This suite includes programs with row- and column-level policies of varying granularity levels, and those necessitating the use of SMT solvers for verification. Furthermore, the tests verify the behaviour of the dependency analysis by incorporating complex conditionals, loops, and implicit and explicit outputs. The tests can be found in the implementation repository [23].

### C. Use cases

We evaluate DiVerT on four use cases inspired by real-world problems in which disjunctive policies naturally arise. The purpose of this evaluation is to validate the security analysis of DiVerT on realistic scenarios involving disjunctive policies, and ensure that its behaviour is consistent with the definitions of Section IV-B. Rather than analysing complete applications for each example, we therefore focus on smaller kernels that capture the core security-critical behaviour of the respective problem.

**Privacy-preserving location service.** Multilateration is a technique to determine the location of a user by measuring their distance to known reference points [25]. Two distances are sufficient to narrow a user's location down to one of two points on a map, and three identify the location unambiguously. Consider a location service provider which tracks, for some number of users, not only their precise location but also their distances to certain points of interest (PoI) such as restaurants or shops. An advertiser wants to query this service to provide location-based ads. For example, if the user is close to a shop $A$, and $A$ has a sale going on, the user may be enticed by this information.

Privacy and business considerations make it desirable to not reveal the precise location of the user to the advertisement company accessing the database, while still allowing for some location-based services in this vein. If the advertiser were to learn the distance of a single user to two or more PoIs at a specific time, the user's location could be inferred. However, we may still want to release the user's distance to any one PoI which they are currently closest to. This can be interpreted as a disjunctive policy, in which the information revealed for each user is bounded by the disjunction of that user's distances to some *single* PoI.

The database schema consists of a single table `Distance(id, poi, dis, loc)`, which stores the ID of each user, the name of the PoI, their distance, and the user's precise location. We implement a small example with two PoIs $\{$'restaurant', 'mall'$\}$ and two users $\{1, 2\}$. Let the view $v_{i,j}$ for each user $i$ and PoI $j$ be defined as the query `SELECT id, poi FROM Distance WHERE id =` $i$ `AND poi =` $j$. The disjunctive policy then covers every combination of user and PoI as a possibility: $\{\{v_{1,\text{'restaurant'}}, v_{2,\text{'restaurant'}}\}, \{v_{1,\text{'restaurant'}}, v_{2,\text{'mall'}}\}, \{v_{1,\text{'mall'}}, v_{2,\text{'restaurant'}}\}, \{v_{1,\text{'mall'}}, v_{2,\text{'mall'}}\}\}$.

We test two programs against this policy. In one, the advertiser uses internal parameters identifying a target user and interest, and issues a single query requesting that user's distance from the relevant point of interest. In the other, the advertiser still targets a particular user, but queries all of that user's distances. As expected, DiVerT accepts the former program, but rejects the latter.

**Privacy-preserving data publishing.** Expanding upon the motivating example in the introduction, we consider the case of programs querying a database with personally identifiable information (i.e., quasi-identifiers). As discussed before, revealing too many quasi-identifiers may make it possible to identify an individual. We consider the example of a medical database [5] with a table `Patients(zip, gen, dis)` storing the ZIP code of residence, gender and disease of patients. An agent querying the database should not learn more than two of these at a time. For simplicity's sake, we only consider queries that retrieve the same data from each patient. Defining $v_1$ = `SELECT dis, gen FROM Patients`, $v_2$ = `SELECT zip, gen FROM Patients`, and $v_3$ = `SELECT zip, dis FROM Patients`, the disjunctive policy can then be written as $\{\{v_1\}, \{v_2\}, \{v_3\}\}$.

Once again, we validate two programs against this policy. Branching on an internal parameter, the client will issue one query to select data for either male or female pa-

tients. In the first program, all queries take the form of `SELECT dis FROM Patients WHERE gen = 'F'`, whereas in the second one, one of the queries additionally filters on the ZIP code: `SELECT dis FROM Patients WHERE gen = 'F' AND zip =` 10001. Again, only the latter program is rejected by DIVERT. This reveals a potential subtlety, as data dependency and hence release of information may arise not only from what columns are selected, but also from conditions restricting the set of rows.

**Secret sharing.** We implement a $(t, n)$ secret sharing schema that splits a secret value $s$ into $n$ shares $s_1, s_2, ..., s_n$. These shares are then distributed among $n$ parties $p_1, p_2, ..., p_n$, each receiving a unique share. A secure secret sharing schema requires that the secret $s$ can only be reconstructed if $t$ or more participants combine their shares. If the number of combined shares is less than $t$, no information about the secret should be revealed. This requirement naturally translates to a disjunctive policy $s_1 \vee s_2 \vee ... \vee s_n$, stipulating that participants can each only learn *one* share.

We assume that the shares $s_1, s_2, ..., s_n$ are created by a secure secret sharing schema and are then stored in a database. The database schema consists of the table `Shares(shareID, shareVal)` which stores the ID of each share and their corresponding value.

The policy only allows a user to read one of the shares (i.e., only one row of the table). We define the view $v_i$ for each share as `SELECT shareVal, shareID FROM Shares WHERE shareID =` $i$ where $i = 1, ..., n$. The corresponding disjunctive policy is going to look like $\{\{v_1\}, \{v_2\}, ..., \{v_n\}\}$.

We implement a program that executes a subroutine for each user, issuing a database query to retrieve the user's share. For example the query for a user to retrieve the share number 5 is `SELECT shareVal FROM Shares WHERE shareID =` 5 and it is correctly accepted by DIVERT. If the same user issues another query to retrieve share number 6, it violates the policy and hence the program is rejected. This scenario shows that DIVERT is able to correctly enforce row-level policies precisely.

**Online shop.** This use case models an online shop and a user with a gift card can only use it to "buy" items that match the value of the gift card. Here we consider a scenario with an online shop that only provides digital items and they are stored in a database. The database schema consists of the items table `Items(id, name, data)` which stores the ID and name of each digital item. We define a view $v_n$ for each item as `SELECT data, name FROM Items WHERE name =` $n$ where $n$ is the item's name.

Assume a database that has the items *Movie*, *CinemaTicket*, *Audiobook*, *Ebook*, and *GymMem*. A policy should only allow the user to access a certain amount of items whose value adds up to value of gift card. For instance a disjunctive policy may look like: $\{\{v_{\text{Movie}}, v_{\text{CinemaTicket}}\}, \{v_{\text{Audiobook}}, v_{\text{Ebook}}\}, \{v_{\text{GymMem}}\}, \{v_{\text{CinemaTicket}}, v_{\text{Ebook}}\}\}$.

We model a user program that issues queries to select items, e.g., `SELECT data FROM Items WHERE name = 'Movie'`.

DIVERT accepts this query because view $v_{\text{Movie}}$ allows the user to access *Movie*. We create two different scenarios; in one the user issues another query asking for *Audiobook*, which DIVERT rejects. In the second scenario, the user asks for *CinemaTicket* which is allowed by the policy, and hence DIVERT accepts it.

## VII. Related Work

This section puts our contributions in the context of related works in the areas of information flow security and database security, discussing security models of dependencies and tractable enforcement mechanisms. To our knowledge, we are the first to explore enforcement mechanisms for disjunctive policies, as well as to reconcile semantic models of (disjunctive) dependencies across the areas of information flow control and database access control.

**Security models.** Semantic models of dependencies have a long history since the introduction of the Lattice of Information (LoI) by Landauer and Redmond [7]. These models define a lattice structure to represent information as equivalence relations ordered by refinement and serve as cornerstone to justify soundness of various dependency analysis at the heart of enforcement mechanisms for security. For example, the universal lattice by Hunt and Sands [26] models dependencies between program variables such that the lattice elements are sets of variables ordered by set containment, and uses it to justify soundness against baseline security conditions, e.g., noninterference [27].

Within the database community, Bender et al. [4], [8] define the notion of Disclosure Lattice to represent the information disclosed by sets of database queries. Disclosure Lattice has been further developed by Guarnieri et al. [14] to enforce conjunctive information-flow policies for database-backed programs. We point out that not all disclosure orders are suitable to represent information disclosure in the context of information flow control: By studying its relation to LoI, we show that query determinacy and the stronger notion of equivalent query rewriting [20] provide sound abstraction, while query containment does not.

Our work builds on recent work by Hunt and Sands [9], which provides a semantic model for disjunctive dependencies, under the notion of the Quantale of Information. We study quantale structures in the context of databases, providing support for disjunctive policies in database-backed programs. While these policies are rooted in the area of access control, cf. ethical wall policies [28], the work of Hunt and Sands [9] is the first to provide an extensional characterization as information-flow policies. Drawing on our new notion of Determinacy Quantale, we develop a security condition to capture the security of database-backed programs in presence of disjunctive database policies.

**Enforcement mechanisms.** The problem of enforcing disjunctive policies for programs and/or databases is completely

unexplored. We study how a standard type-based program analysis [15], equipped the notion of path sensitivity, can be adapted to statically capture disjunctive program dependencies.

At the core of our analysis is a new abstraction of database queries which enables flexible enforcement of disjunctive policies by means of SMT solvers, as witnessed by our use cases. An immediate benefit of our Determinacy Quantale is that we can prove soundness of the enforcement with respect to a solid semantic baseline for disjunctive dependencies.

There exists a wide array of works enforcing conjunctive policies for database-backed programs. Guarnieri et al. [14] propose dynamic monitoring to enforce database policies. Their abstractions are limited to boolean queries and rely on the Disclosure Lattice of Bender et al. [4], [8], which may cause soundness issues when assuming query containment as the underlying lattice order.

Language-integrated queries are supported by a range of works such as SIF [10] and JSLINQ [12], SELINKS [11], UrFlow [29], DAISY [14], Jacqueline [30], and LWeb [13] for row- and column-level conjunctive policies. These works apply PL-based enforcement techniques such as type systems, dependent types, refinement types, and symbolic execution to database-backed programs [13], [14], [31], [32], but lack support for expressing and enforcing disjunctive policies.

Li and Zhang [33] explore path-sensitive program analysis to improve precision of information flow analysis, yet they do not consider disjunctive policies. QAPLA [34] is a database access control middleware supporting complex security policies, such as linking and aggregation policies, with focus only on access control.

## VIII. CONCLUSIONS

We presented a case for the significance of disjunctive dependency analysis to the security of database-backed programs. After reviewing recent theoretical developments in representing disjunctive information, we introduced two structures, the Determinacy Lattice and the Determinacy Quantale, as database-oriented counterparts to theoretical structures representing simple and disjunctive knowledge respectively.

Using these structures, we formulated a security condition which expresses that a database-backed program satisfies a given disjunctive policy. In order to enforce this security condition, we developed a type-based static analysis to compute a bound on the disjunctive dependencies of database-backed programs in a model language. By a series of approximations, this bound itself can be tractably compared to the representation of a static policy.

These steps constitute an enforcement mechanism for disjunctive policies, which we proved sound with respect to our security condition. To showcase this enforcement mechanism, we implemented it in our prototype tool, DIVERT. In order to validate this prototype and the overall framework, we verified the tool on a set of functional tests covering a variety of language features and disjunctive information patterns, as well as several use cases representing real-world scenarios in which we want to enforce disjunctive policies.

## IX. ACKNOWLEDGEMENTS

We are grateful to David Sands and Roberto Guanciale for fruitful discussions, and would also like to thank the anonymous reviewers for their insightful comments and feedback.

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, the Swedish Research Council (VR), and the Swedish Foundation for Strategic Research (SSF).

## REFERENCES

[1] E. Bertino and R. Sandhu, "Database security-concepts, approaches, and challenges," *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 1, pp. 2–19, 2005.

[2] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[3] M. Guarnieri, S. Marinovic, and D. Basin, "Strong and provably secure database access control," in *IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 163–178.

[4] G. Bender, L. Kot, and J. Gehrke, "Explainable security for relational databases," in *ACM SIGMOD International Conference on Management of data*. ACM, 2014, pp. 1411–1422.

[5] L. Sweeney, "k-anonymity: A model for protecting privacy," *International Journal of Uncertainty, Fuzziness and Knowledge-based Systems*, vol. 10, no. 05, pp. 557–570, 2002.

[6] C. Dwork, "Differential privacy," in *Automata, Languages and Programming: 33rd International Colloquium*. Springer, 2006, pp. 1–12.

[7] J. Landauer and T. Redmond, "A lattice of information," in *Proceedings Computer Security Foundations Workshop*. IEEE, 1993, pp. 65–70.

[8] G. M. Bender, L. Kot, J. Gehrke, and C. Koch, "Fine-grained disclosure control for app ecosystems," in *ACM SIGMOD International Conference on Management of Data*. ACM, 2013, pp. 869–880.

[9] S. Hunt and D. Sands, "A quantale of information," in *IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE, 2021, pp. 1–15.

[10] S. Chong, K. Vikram, A. C. Myers *et al.*, "Sif: Enforcing confidentiality and integrity in web applications," in *16th USENIX Security Symposium (USENIX Security)*. USENIX Association, 2007, pp. 1–16.

[11] B. J. Corcoran, N. Swamy, and M. Hicks, "Cross-tier, label-based security enforcement for web applications," in *ACM SIGMOD International Conference on Management of data*. ACM, 2009, pp. 269–282.

[12] M. Balliu, B. Liebe, D. Schoepe, and A. Sabelfeld, "Jslinq: Building secure applications across tiers," in *6th ACM Conference on Data and Application Security and Privacy*. ACM, 2016, pp. 307–318.

[13] J. Parker, N. Vazou, and M. Hicks, "Lweb: Information flow security for multi-tier web applications," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, 2019.

[14] M. Guarnieri, M. Balliu, D. Schoepe, D. Basin, and A. Sabelfeld, "Information-flow control for database-backed applications," in *IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 79–94.

[15] B. v. Delft, S. Hunt, and D. Sands, "Very static enforcement of dynamic policies," in *International Conference on Principles of Security and Trust*. Springer, 2015, pp. 32–52.

[16] A. M. Ahmadian, M. Soloviev, and M. Balliu, "Disjunctive policies for database-backed programs," 2023. [Online]. Available: https://arxiv.org/abs/2312.10441

[17] I. Kaplansky, *Set Theory and Metric Spaces*. AMS Chelsea Publishing, 2001.

[18] B. A. Davey and H. A. Priestley, *Introduction to lattices and order*. Cambridge University Press, 2002.

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

[20] A. Nash, L. Segoufin, and V. Vianu, "Views and queries: Determinacy and rewriting," *ACM Transactions on Database Systems (TODS)*, vol. 35, no. 3, pp. 1–41, 2010.

[21] Q. Wang and K. Yi, "Conjunctive queries with comparisons," in *International Conference on Management of Data*. ACM, 2022, pp. 108–121.

[22] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, "Termination-insensitive noninterference leaks more than just a bit," in *13th European Symposium on Research in Computer Security*. Springer, 2008, pp. 333–348.
[23] A. M. Ahmadian, M. Soloviev, and M. Balliu, "Divert," 2023, software release. [Online]. Available: https://github.com/KTH-LangSec/DiVerT
[24] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2008, pp. 337–340.
[25] W. Murphy and W. Hereman, "Determination of a position in three dimensions using trilateration and approximate distances," *Department of Mathematical and Computer Sciences, Colorado School of Mines, Golden, Colorado, MCS-95*, vol. 7, p. 19, 1995.
[26] S. Hunt and D. Sands, "On flow-sensitive security types," in *33rd SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL*. ACM, 2006, pp. 79–90.
[27] J. A. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 1982, pp. 11–20.
[28] D. F. Brewer and M. J. Nash, "The chinese wall security policy." in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 1989, pp. 206–214.
[29] A. Chlipala, "Static checking of dynamically-varying security policies in database-backed applications," in *9th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2010, pp. 105–118.
[30] J. Yang, T. Hance, T. H. Austin, A. Solar-Lezama, C. Flanagan, and S. Chong, "Precise, dynamic information flow for database-backed applications," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 631–647, 2016.
[31] N. Swamy, B. J. Corcoran, and M. Hicks, "Fable: A language for enforcing user-defined security policies," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2008, pp. 369–383.
[32] L. Lourenço and L. Caires, "Dependent information flow types," in *42nd SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL*. ACM, 2015, pp. 317–328.
[33] P. Li and D. Zhang, "Towards a flow- and path-sensitive information flow analysis," in *30th IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 2017, pp. 53–67.
[34] A. Mehta, E. Elnikety, K. Harvey, D. Garg, and P. Druschel, "Qapla: Policy compliance for database-backed systems," in *26th USENIX Security Symposium (USENIX Security)*. USENIX Association, 2017, pp. 1463–1479.