Information-Flow Control for Database-backed Applications

Marco Guarnieri*, Musard Balliu†, Daniel Schoepe‡, David Basin§, and Andrei Sabelfeld‡
*IMDEA Software Institute †KTH Royal Institute of Technology ‡Chalmers University of Technology §ETH Zurich

***Abstract*—Securing database-backed applications requires tracking information across the application program and the database together, since securing each component in isolation may still result in an overall insecure system. Current research extends language-based techniques with models capturing the database's behavior. This research, however, relies on simplistic database models, which ignore security-relevant features that may leak sensitive information.**

We propose a novel security monitor for database-backed applications. Our monitor tracks fine-grained dependencies between variables and database tuples by leveraging database theory concepts like disclosure lattices and query determinacy. It also accounts for a realistic database model that supports security-critical constructs like triggers and dynamic policies. The monitor automatically synthesizes program-level code that replicates the behavior of database features like triggers, thereby tracking information flows inside the database. We also introduce symbolic tuples, an efficient approximation of dependency-tracking over disclosure lattices. We implement our monitor for SCALA programs and demonstrate its effectiveness on four case studies.

I. INTRODUCTION

Database-backed applications are programs that interact with databases to store and retrieve information. These applications are commonly used in settings like e-commerce, e-health, and social networks, and often handle sensitive data where security is a concern.

Securing database-backed applications is challenging: the security of the program and the database in isolation is insufficient to ensure the overall system's security. For instance, program-level information, such as the sensitive context of a function call that triggers a query, is lost at the time of database-level enforcement. Conversely, database-level information, such as fine-grained security labels, is lost at the time of program-level enforcement, when information from the database is manipulated by the application.

Security models for database-backed applications must therefore account for both the program's and the database's semantics. Following this approach, existing information-flow control (IFC) solutions [7], [14], [15], [17], [19], [31], [44], [49] extend programs with database models and apply standard IFC techniques, such as security type systems [17], [43], symbolic execution [14], or faceted values [49], to track information flows across the program and the database, with the goal of providing end-to-end security.

These approaches, however, are inadequate to secure modern database-backed applications. They only consider simplistic database models and often ignore features like dynamic policies and triggers. These features are available in most modern database systems and can be exploited to violate the database's confidentiality [25]. Ignoring them, therefore, means ignoring possible information leaks.

Another challenge in tracking information flows across the program-database boundary is analyzing queries. Some approaches [7], [43] perform simple syntactic checks on table and column identifiers to derive the queries' security levels. As modern query languages like SQL are very expressive, this may result in coarse approximations that make the analyses imprecise. Additionally, these approaches do not support common policy idioms used in database security, such as row-level policies.

In summary, effectively securing database-backed applications requires (1) realistic database models that capture the security-critical features offered by modern databases, and (2) specialized techniques, rooted in database theory, to analyze queries.

Contributions. We develop a novel IFC solution that (1) builds on top of a realistic database model accounting for a large class of security-relevant features, and (2) tracks fine-grained dependencies between variables and tuples by using concepts from database theory.

First, we develop a foundation for IFC for database-backed applications using WHILESQL, a simple imperative language extended with querying capabilities. WHILESQL builds on a state-of-the-art database operational semantics developed by Guarnieri et al. [25] and supports database features like triggers, views, and dynamic policies. We propose a novel security condition for WHILESQL programs that accounts for dynamic policy changes.

Second, we develop a novel IFC monitor for WHILESQL programs and prove it sound with respect to our security condition. Our monitor tracks fine-grained dependencies between variables and queries across program-level computations and blocks outputs that could potentially leak sensitive information. For checking policy violations, the monitor relies on disclosure lattices [8] and query determinacy [35]. The monitor supports row-level policies, a common class of database policies used in many fine-grained access control models [12], [24], [37], [48]. Additionally, it supports security-critical database features, such as triggers and policy changes, that are not supported by existing mechanisms [17], [19], [31], [43], [44], [49]. To address the mismatch between program code and database features like triggers and integrity constraints, the monitor automatically synthesizes WHILESQL code mimicking these features' behavior, thereby enabling IFC techniques to track information flows inside the database.

Third, we implement our approach in DAISY (DAtabase and Information-flow SecuritY), a security monitor for database-backed SCALA programs. To overcome undecidability issues when reasoning with disclosure lattices, DAISY relies on symbolic tuples, a novel, efficient approximation of dependency-tracking over disclosure lattices. We demonstrate our approach's precision and feasibility in four case studies implementing (i) a social network, (ii) an assignment grading system, (iii) a calendar application, and (iv) a conference-management system. The case studies confirm that DAISY successfully prevents leaks of sensitive information in the presence of realistic database constructs without being overly restrictive. Our experiments also show that symbolic tuples can be used to efficiently track fine-grained dependencies. Concretely, DAISY introduces an overhead of only 5%–10% in our case studies.

II. OVERVIEW

We now present our approach via an example. First, we introduce the system model and the setting of our example. Next, we motivate the need for realistic database models for IFC. Finally, we illustrate how our monitor DAISY prevents leaks of sensitive information.

System model. The system consists of *users*, whose interaction with the database is mediated by a program like a web application. Each user is uniquely associated with a user account that is used to authenticate the user and retrieve information from the database. We assume that users execute programs using their own accounts. An *attacker* is a user who can interact with the database only through programs. He cannot learn the results of the queries issued by the program unless they are part of the program's output.

A security policy is defined at the database level using access control policies, which specify what data each user is allowed to access. Differently from access control, however, we interpret the *read* permissions over tables and views as *information-flow policies*, and we enforce them in an end-to-end fashion across the program and the database. We assume that the database does not enforce *read* permissions over tables and views, but it still correctly enforces *write* permissions, e.g., a user can insert a tuple into a table T only if the policy says so. This allows us to study what it means for a system to be end-to-end secure from the information-flow perspective.

Setting. We consider a *social network* allowing users to review books, publish their reviews, and share them with friends. The database consists of six tables: `book`, `user`, `friends`, `review`, `likes`, and `stats`. The table `book` contains information about books, the table `user` contains the users' information, the table `friends` encodes the friendship relation among users, the table `review` contains the users' reviews, the table `likes` stores information about reviews liked by users, and the table `stats` contains statistics about the users and reviews. Furthermore, we assume that for each user u there is a database view $\texttt{review}_u$ containing user u's reviews, i.e., the results of the query `SELECT * FROM review WHERE userId` $= u$.

The security policy is as follows: all users can read the content of the tables `book`, `user`, `friends`, `likes`, and `stats` but they can only read their friends' reviews. The first requirement can be implemented by granting `SELECT` permissions over the respective tables. The second requirement is formalized using row-level policies, which disclose only a subset of the tuples in a table. Row-level policies are a widely used policy idiom in database security, and they are employed in many fine-grained database access control models [12], [24], [37], [48]. In our setting, we model the second requirement by granting `SELECT` permissions over the view $\texttt{review}_{u_1}$ to u_2 whenever $\langle u_1, u_2 \rangle$ is in the table `friends`. We remark that we interpret the above policy as an information-flow policy, not as an access control one.

Motivating example. We consider three users $Alice$, Bob, and $Carl$. We assume that $Alice$ is a friend of Bob and $Carl$, but Bob and $Carl$ are not friends with each other. That is, $Alice$ can read Bob's and $Carl$'s reviews, but Bob cannot read $Carl$'s reviews and vice versa.

Consider the simple program below. First, $Carl$ reviews the novel "War and Peace" by Leo Tolstoy. Next, $Alice$ reads Carl's review, which she appreciates, and creates an entry in the table `likes` associated with it. Finally, Bob retrieves from `stats` the statistics of all his friends.

```
//Executed by Carl
x ← INSERT INTO review(id, user, book, score)
    VALUES (1, Carl, "War and Peace", 10)
//Executed by Alice
y ← SELECT revId, text, score FROM review WHERE
    book = "War and Peace" AND userID = Carl
out(Alice, y)
z ← INSERT INTO likes VALUES (y.revId,
    "War and Peace", Carl, Alice)
//Executed by Bob
F ← SELECT u2 FROM friends WHERE u1 = Bob
S ← SELECT genre FROM stats WHERE userId = Bob
for (f : F; g : S)
  v ← SELECT v FROM stats WHERE userId = f
        AND genre = g
  out(Bob, ⟨f, g, v⟩)
```

The program is secure since all information flows comply with the policy. Specifically, $Alice$ observes one of $Carl$'s reviews. This is allowed by the policy since they are friends. Moreover, Bob's computation depends only on the public tables `friends` and `stats`.

Why are realistic database models essential? The above example relies on only basic database features like `SELECT` and `INSERT` commands. Modern databases, however, support many security-critical features, such as dynamic policies and triggers, that may introduce additional information flows. As a result, a seemingly secure program may actually be insecure when features like triggers are accounted for.

To illustrate this, we extend our social network with a trigger, that is, SQL code that is executed automatically by the database in response to queries. Concretely, our social network collects several statistics about users' reviews in the table `stats`. Among other things, the social network collects, for each user u and genre g, the `score` of the last review of books

of genre g liked by u. Instead of computing this data on the fly, the statistics are stored in the database and updated using triggers. The following trigger, which is executed under the database administrator's privileges, updates the score whenever a new tuple is inserted into the table `likes`.

```
CREATE TRIGGER tr ON likes AFTER INSERT DO
  UPDATE stats SET lastScore = (SELECT score
    FROM reviews WHERE id = NEW.revid)
  WHERE user = NEW.user AND genre IN (SELECT
    genreFROM book WHERE book = NEW.book)
```

Specifically, whenever someone inserts a tuple $\langle revId, book, revAuthor, user\rangle$ into `likes`, the trigger updates the score associated with the user $user$ and $book$'s genre with the score associated with the review with identifier $revId$. In the above trigger, we write `NEW.`x to refer to the attribute x of the tuple just inserted in `likes`.

The program is no longer secure when the trigger tr is present in the database. Indeed, now the information observed by Bob depends on $Carl$'s review. This flow of information, however, is not allowed by our security policy since Bob can only read his friends' reviews. In more detail, when $Alice$ inserts the tuple into the table `likes`, the trigger tr is executed and the attribute `lastScore` is updated using the score in $Carl$'s review. Moreover, since $Carl$ is one of $Alice$'s friends, this information influences Bob's computation, thereby violating the security policy.

Stopping leaks with DAISY. Ignoring advanced database features may lead to a false sense of security. Indeed, a seemingly secure program may still leak sensitive information due to additional information flows introduced by triggers and other database features. As a result, reasoning about the security of database-backed applications requires accounting for realistic database models and for common policy idioms used in database security. Unfortunately, existing solutions [7], [14], [15], [17], [19], [31], [43], [44], [49] either ignore relevant security-critical database features (like triggers and dynamic policies) or adopt imprecise analyses when handling queries (cf. §VIII). This severely limits their ability to secure applications and to enforce natural policy idioms like row-level policies. To address this, we propose DAISY, a security monitor that leverages disclosure lattices and query determinacy to track fine-grained tuple-level dependencies. DAISY monitors the program's execution, tracks dependencies between variables and tuples, and stops the program whenever sensitive information may be leaked.

How DAISY works DAISY tracks, at runtime, dependencies between queries and program variables and stops the program whenever it detects a possible leak of sensitive information. For instance, whenever information is retrieved from the database, DAISY determines which tuples may have influenced the query's result and it tracks how the retrieved information flows through the program. To concisely represent sets of tuples, we develop *symbolic tuples*, an efficient approximation of disclosure lattices (cf. §VI), which represent sets of concrete tuples using logical formulae.

Consider the program from our example. When $Alice$ retrieves the review, DAISY records that the content of the variable y depends on $Carl$'s review. More precisely, DAISY labels y with the symbolic tuple $\langle$`review, userId` $= Carl \wedge$ `book = "War and Peace"`$\rangle$, which denotes that y's content depends on the values of all tuples in the table `review` satisfying the constraint `userId` $= Carl \wedge$ `book = "War and Peace"`. When $Alice$ inserts a tuple into the table `likes`, DAISY tracks the information flow caused by the trigger. DAISY determines that the `UPDATE` command executed by the trigger inserts sensitive information, i.e., the score of $Carl$'s review, into the public table `stats`. Concretely, the tool compares the label associated with the input values, i.e., the tuple $\langle y.revId$, `"War and Pace"`$, Carl, Alice\rangle$, with the label associated with the table $stats$.

Among others, $\langle y.revId$, `"War and Pace"`$, Carl, Alice\rangle$ is labelled with the symbolic tuple $\langle$`review, userId` $= Carl \wedge$ `book = "War and Peace"`$\rangle$ Using query determinacy, DAISY checks if the symbolic tuple $\langle$`review, userId` $= Carl \wedge$ `book = "War and Peace"`$\rangle$ can be derived from those associated with the `stats` table. Since the `stats` table contains only public information, there is no symbolic tuple among `stats`'s labels that discloses the information represented by $\langle y.revId$, `"War and Pace"`$, Carl, Alice\rangle$'s label $\langle$`review, userId` $= Carl \wedge$ `book = "War and Peace"`$\rangle$. Hence, DAISY stops the program, thereby preventing the leak of sensitive information.

Organization. We formalize WHILESQL in §III and our security condition in §IV. We present our monitor in §V and symbolic tuples in §VI. We present DAISY and our case studies in §VII, we discuss related work in §VIII, and we draw conclusions in §IX. A technical report with complete proofs of all results is available at [23], and DAISY is available at [22].

III. WHILESQL

Here we present WHILESQL, a language supporting querying constructs and a realistic database model.

A. Syntax and notation

Syntax. WHILESQL is an imperative language with querying capabilities, whose syntax is given in Figure 1. Its imperative fragment consists of assignments $x := e$, conditionals **if** e **then** c_1 **else** c_2, loops **while** e **do** c, and output statements **out**(u, e), which print the value of an expression e to a user u. Expressions e are values $n \in Val$, variables $x \in Var$, or application of unary $\oslash e$ and binary operations $e_1 \otimes e_2$ to expressions. The set $\mathcal{U}$ of all users is $UID \cup \{public\}$, where UID is a set of user identifiers and $public$ is a designated identifier denoting all users.

Database queries are modeled as statements of the form $x \leftarrow q$ that execute an SQL command q, which may contain program variables, and assign the result to a variable x. Observe that each SQL command either returns the query's result or an error message. Error messages indicate whether queries violate security constraints or integrity constraints, such as a `DELETE` command that is not allowed by the current security policy or

Basic Types		**Syntax**	
(Table Ids)	$T \in \mathbb{T}$	*(Privileges)*	$p :=$ `SELECT ON` R | `INSERT ON` T | `DELETE ON` T
(View Ids)	$V \in \mathbb{V}$		| `CREATE VIEW` | `CREATE TRIGGER ON` T
(Relation Ids)	$R \in \mathbb{T} \cup \mathbb{V}$	*(Actions)*	$a :=$ `INSERT` $e_1, \ldots, e_n$ `INTO` T | `DELETE` $e_1, \ldots, e_n$ `FROM` T
(Trigger Ids)	$tr \in \mathbb{TR}$		| `GRANT` p `TO` u | `REVOKE` p `FROM` u
(Variables)	$x \in Var$		| `GRANT` p `TO` u `WITH GRANT OPTION`
(Values)	$n \in Val$	*(SQL commands)*	$q := a$ | `SELECT` φ | `CREATE VIEW` V `:` `SELECT` φ
(User identifiers)	$u \in \mathcal{U}$		| `CREATE TRIGGER` tr `ON` T `AFTER` (`INS` | `DEL`) `IF` φ `DO` a
(Formulae)	$\varphi \in RC$	*(Expressions)*	$e := n \mid x \mid \oslash e_1 \mid e_1 \otimes e_2$
		(Statements)	$c := \varepsilon \mid x \leftarrow q \mid x := e \mid$ **out**(u, e) | **if** e **then** c_1 **else** c_2
			| **while** e **do** $c \mid c_1 \,;\, c_2$

Fig. 1: WHILESQL's syntax

an `INSERT` command that violates a primary key constraint. WHILESQL supports SQL's core features, such as `SELECT`, `INSERT`, `DELETE`, `GRANT`, and `REVOKE` commands, as well as advanced features like triggers and views.

Database features. WHILESQL relies on the state-of-the-art database semantics from Guarnieri et al. [25], which supports security-critical features like dynamic policies and triggers. Hence, following [25], we make various simplifications to our query language.

WHILESQL supports retrieving information from the database using `SELECT` commands. Rather than using SQL's data query language, we rely on the relational calculus (i.e., function-free first-order logic), which has a simple and well-defined semantics [1]. Following [25], we only consider boolean queries, i.e., queries whose results are either `true` or `false`. We denote by RC the set of all boolean relational calculus queries.

WHILESQL allows changes to the database's content using `INSERT` and `DELETE` commands. Specifically, we support `INSERT` and `DELETE` commands that explicitly identify the tuple to be inserted or deleted, i.e., commands of the form `INSERT INTO` $table(x_1, \ldots, x_n)$ `VALUES` $(v_1, \ldots, v_n)$ and `DELETE FROM` $table$ `WHERE` $x_1 = v_1 \wedge \ldots \wedge x_n = v_n$, where $x_1, \ldots, x_n$ are $table$'s attributes and $v_1, \ldots, v_n$ are the tuple's values. More complex commands can be simulated by combining `SELECT`, `INSERT`, and `DELETE` commands.

WHILESQL also supports the administration of dynamically changing security policies. We support `GRANT` commands to add permissions to a security policy. We also support delegation through `GRANT` commands with `GRANT OPTION`. Moreover, privileges can be revoked using `REVOKE` commands. We only consider `REVOKE` commands with the `CASCADE OPTION`, i.e., when a user revokes a privilege, he also revokes all the privileges that depend on it [40], [47].

Our model also supports triggers, which are procedures automatically executed by the database system in response to user commands. In particular, we support `AFTER` triggers on `INSERT` and `DELETE` events, i.e., triggers that are executed in response to `INSERT` and `DELETE` commands. In our model, triggers are executed under the privileges of the trigger's owner. Moreover, the triggers' `WHEN` conditions (which specify whether a trigger is enabled or not) are arbitrary boolean queries and their actions are `INSERT` or `DELETE` commands. Note that database systems usually impose restrictions on the `WHEN` clause, such as it must not contain sub-queries. However, most systems can express arbitrary conditions on triggers by combining control flow statements with `SELECT` commands inside the trigger's body. Thus, we support the class of triggers whose body is of the form `BEGIN IF` $expr$ `THEN` act `END`, where $expr$ is a boolean query and act is an `INSERT` or `DELETE` command. Following [25], we only consider triggers that do not recursively activate other triggers.

We also support database views, i.e., virtual tables defined through `SELECT` queries, executed under the privileges of the view's owner. Additionally, we support `CREATE` commands for creating new triggers and views. Finally, we support two kinds of integrity constraints: functional dependencies and inclusion dependencies [1]. They model the most widely used SQL integrity constraints, i.e., the `UNIQUE`, `PRIMARY KEY`, and `FOREIGN KEY` constraints.

B. Local semantics

We define here the semantics of WHILESQL programs executed in isolation by a user u. It is formalized as a ternary relation $\langle c, m, s\rangle \xrightarrow{o}_u \langle c', m', s'\rangle$ mapping a local configuration $\langle c, m, s\rangle$, where c is the program under execution, m is the memory, and s is the database state, to a configuration $\langle c', m', s'\rangle$ while producing an observation o.

A WHILESQL program is defined with respect to a *database configuration* $\langle D, \Gamma\rangle$, where D is a database schema, i.e., a set of table identifiers with the corresponding arities, and Γ is a set of integrity constraints. Here, we fix a database configuration $M = \langle D, \Gamma\rangle$.

Database states. Following [25], we now introduce all the components necessary to model a database state.

We define a *security policy* to be a finite set of `GRANT` statements. Given a policy sec and a user u, $auth(sec, u)$ denotes the set of all tables and views that u is authorized to read according to sec. A *system state* is a tuple $\langle db, U, sec, T, V\rangle$, where db is a database state, $U \subset UID$ is a finite set of users, sec is a security policy, T is a finite set of triggers, and V is a finite set of views. We lift $auth$ from policies to system states, i.e., $auth(\langle db, U, sec, T, V\rangle, u) = auth(sec, u)$.

A *context* ctx describes the database's history, the scheduled triggers that must be executed, and how to modify the database's state in case a roll-back occurs. We refer the reader to [25] for a formal definition of contexts. A *runtime state* is a tuple $\langle s, ctx\rangle$, where s is a system state and ctx is a context. The set of all runtime states is denoted by Ω_M and ϵ denotes the empty context. In the following, we use s to refer to both system and runtime states when this is clear from the context, and we use $\langle s, ctx\rangle$ otherwise.

Local configurations. A *local configuration* $\langle c, m, \langle s, ctx\rangle\rangle$ consists of a command $c \in Com$, a memory $m \in Mem$, and a runtime state $\langle s, ctx\rangle \in \Omega_M$, where *memories* $m \in Mem$ are functions mapping variables to values, i.e., $Mem = Var \to Val$. A configuration is *initial* iff $ctx = \epsilon$.

Observations. In WHILESQL, there are two ways of producing observations. First, $\mathbf{out}(u, e)$ statements can be used to output information to users. Second, successfully executed `GRANT`, `REVOKE`, and `CREATE` commands produce public observations notifying all users of the configuration's changes. Formally, an *observation* is a tuple $\langle u, o\rangle$, where $u \in \mathcal{U}$ is the target user and o is a value in Val or a `GRANT`, `REVOKE`, or `CREATE` command. We denote by Obs the set of all observations.

In our model, we represent traces of observations using sequences, for which we use a standard notation. For a set S, S^* is the set of all finite sequences over S. Given a sequence $s \in S^*$, we denote by $|s|$ its length, by s^j, where $j \in \mathbb{N}$, its prefix of length j, and by $s|_j$ its j-th element (if it exists). We also denote by ε the empty sequence, by $s_1 \cdot s_2$ the concatenation of s_1 and s_2, and by $s_1 \preceq s_2$ that s_1 is a prefix of s_2.

Evaluation relation. Given a user $u \in UID$, the relation $\to_u \subseteq (Com \times Mem \times \Omega_M) \times Obs \times (Com \times Mem \times \Omega_M)$ formalizes the local operational semantics of programs executed by u. A *run* r is an alternating sequence of configurations and observations that starts with an initial configuration and respects the rules defining $\to_u$. Given a run r, we denote by r^i, where $i \in \mathbb{N}$, the run obtained by truncating r at the i-th state. A *trace* is an element of Obs^*. The trace τ of a run r, denoted by $trace(r)$, is obtained by concatenating all observations in the run.

We rely on [25] for the semantics of SQL statements. Our operational semantics uses the function $[\![q]\!](\langle s, ctx\rangle, u)$ (defined in [23]) to connect WHILESQL's semantics with the database's semantics. The function $[\![q]\!](\langle s, ctx\rangle, u)$ takes as input an SQL command q, a runtime state $\langle s, ctx\rangle \in \Omega_M$, and the user $u \in UID$ executing the command, and it returns a tuple $\langle\langle s', ctx'\rangle, r, em\rangle$, where $\langle s', ctx'\rangle \in \Omega_M$ is the new runtime state, r is q's result, and em is an error message. We also write $[\![e]\!](m)$ to denote the evaluation of an expression e in memory m. It is always clear from context if $[\![\cdot]\!](\cdot)$ refers to queries or expressions.

Figure 2 depicts the rules specifying a query's execution. The rule E-QUERYOK handles the successful execution of queries. It first replaces the free variables in the query with their values. Afterwards, it executes the query (using $[\![q]\!](\langle s, ctx\rangle, u)$) and it stores the query's result in the memory. The rule relies on the function $obs(q)$, which takes as input a query q, to conditionally produce a public observation $\langle public, q\rangle$ in case the command

E-QUERYOK

$$\frac{\begin{array}{c}\{v_1, \ldots, v_n\} = vars(q) \\ [\![q']\!](\langle s, ctx\rangle, u) = \langle\langle s', ctx'\rangle, r, \epsilon\rangle \\ q' = q[v_1 \mapsto [\![v_1]\!](m), \ldots, v_n \mapsto [\![v_n]\!](m)]\end{array}}{\langle x \leftarrow q, m, \langle s, ctx\rangle\rangle \xrightarrow{obs(q')}_u \langle \varepsilon, m[x \mapsto r], \langle s', ctx'\rangle\rangle}$$

E-QUERYEX

$$\frac{\begin{array}{c}\{v_1, \ldots, v_n\} = vars(q) \\ [\![q']\!](\langle s, ctx\rangle, u) = \langle\langle s', ctx'\rangle, r, em\rangle \quad em \neq \epsilon \\ q' = q[v_1 \mapsto [\![v_1]\!](m), \ldots, v_n \mapsto [\![v_n]\!](m)]\end{array}}{\langle x \leftarrow q, m, \langle s, ctx\rangle\rangle \to_u \langle \varepsilon, m[x \mapsto em], \langle s', ctx'\rangle\rangle}$$

Fig. 2: Rules handling the query's execution

q modifies the database configuration. Formally, $obs(q) = \langle public, q\rangle$ if q is a `GRANT`, `REVOKE`, or `CREATE` command, and ε otherwise. Hence, the rule guarantees that configuration changes are visible to all users. The rule E-QUERYEX handles queries that fail, e.g., due to an integrity constraint's violation. Instead of storing the query result, the rule stores the error message in the memory. The rules for the other WHILESQL statements are standard and the full details are given in [23].

C. Global semantics

We now introduce a semantics modeling multiple WHILESQL programs executed in parallel. We formalize it as a ternary relation $\langle C, M, s, \mathcal{S}\rangle \xrightarrow{o} \langle C', M', s', \mathcal{S}'\rangle$ mapping a global configuration $\langle C, M, s, \mathcal{S}\rangle$, where C is the sequence of programs under execution, M is the sequence of memories, s is the state of the shared database, and $\mathcal{S}$ is the scheduler's state, to a global configuration $\langle C', M', s', \mathcal{S}'\rangle$, while producing the observation o.

Global configurations. We denote the set of commands together with the executing user by $Com_{UID} = UID \times Com$ and the set of pairs of users and memories as $Mem_{UID} = UID \times Mem$. To model a system state where multiple WHILESQL programs run in parallel and share a common database, we introduce global configurations. A *global configuration* is a tuple $\langle C, M, \langle s, ctx\rangle, \mathcal{S}\rangle \in GlConf$, where $C \in Com^*_{UID}$ is a sequence of WHILESQL programs paired with the executing users, $M \in Mem^*_{UID}$ is a sequence of memories, $\langle s, ctx\rangle \in \Omega_M$ is the runtime state of the shared database, and $\mathcal{S}$ is a scheduler formalizing the interleaving of the programs in C. We consider only configurations $\langle C, M, \langle s, ctx\rangle, \mathcal{S}\rangle$ such that $|C| = |M|$ and for all $1 \leq i \leq |C|$, $C|_i = \langle u, c\rangle$ and $M|_i = \langle u, m\rangle$. Furthermore, a *global state* is a pair $\langle M, s\rangle$, where $M \in Mem^*_{UID}$ and s is a system state.

Evaluation relation. Our global semantics is standard and it executes, at each computation step, one step of the local semantics for the program selected by the scheduler. We formalize the global semantics in [23]. For simplicity, we assume that each user is associated with at most one program and that different programs use disjoint sets of variable identifiers. Moreover, we assume that all expressions are well-typed, and all SQL commands refer to tables in the database schema or previously created views.

IV. Security model

We introduce our security model in terms of the knowledge of a user that observes outputs and public events from a program execution. To ease the presentation, we assume that only the database's content is sensitive, while the initial memory's content is known by all users. This is without loss of generality, since sensitive information can be loaded from the database at the start of the computation. In our technical report [23], we consider the more general case where the memory content can be sensitive.

A. Preliminaries

Database equivalence. Two database states db and db' are equivalent with respect to a set S of tables and views, written $db \approx_S db'$, iff the contents of all tables and views in S are the same in db and db'. For the equivalence of system states, we employ data-indistinguishability from [25]. Informally, two system states s and s' are equivalent for a user u iff the users, policies, triggers, and views in s and s' are the same and the content of the tables and views that u is authorized to read is the same in s and s'. Formally, two system states $s = \langle db, U, sec, T, V\rangle$ and $s' = \langle db', U', sec', T', V'\rangle$ are *u-equivalent*, written $s \approx_u s'$, iff (1) $U = U'$, (2) $sec = sec'$, (3) $T = T'$, (4) $V = V'$, and (5) $db \approx_{auth(sec,u)} db'$. Given a system state s and a user u, we denote by $[s]_{\approx_u}$ the set of all system states that are u-equivalent to s.

Trace equivalence. To formalize equivalence between traces, we first define the *projection of a trace τ for a user u*, written $\tau\restriction_u$. The projection $\tau\restriction_u$ is the sequence of all observations in τ that u can observe, i.e., those observations where the user is either u or $public$.

Two traces τ_1 and τ_2 are *u-equivalent*, written $\tau_1 \sim_u \tau_2$, iff one of the u-projections is the prefix of the other one, i.e., $\tau_1\restriction_u \preceq \tau_2\restriction_u$ or $\tau_2\restriction_u \preceq \tau_1\restriction_u$. We remark that our definition of trace equivalence follows state-of-the-art definitions for dynamic policies, which do not differentiate between divergence and termination [3], [46]. This is in contrast with other works defining trace equivalence as requiring that either both traces are equal or one is a divergence terminated prefix of the other [4], [26].

B. Knowledge

Following [3], [46], we characterize what a user can infer from an execution in terms of his *knowledge*, i.e., the set of system states consistent with his observations.

Definition 1. The *knowledge $K_u(\langle M_0, s_0\rangle, C, \mathcal{S}, \tau)$ of a user u* for a global state $\langle M_0, s_0\rangle$, a sequence of programs C, a scheduler $\mathcal{S}$, and a trace τ is defined as $\{s \mid s \approx_u s_0 \wedge \forall ctx', \tau', C', M', s', \mathcal{S}'.\ (\langle C, M_0, \langle s, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau'}{}^{*} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle \Rightarrow \tau \sim_u \tau')\}$.

A user u's knowledge is the set of initial system states that u considers possible after having observed $\tau\restriction_u$. Thus, a smaller set indicates a more precise knowledge.

Def. 1 is *progress-insensitive* as it ignores information leaks due to the progress of computation, i.e., information that can be inferred solely by observing *how many* outputs the program produces. We achieve this by requiring that any execution starting from a u-equivalent global state only produces traces τ' that are u-equivalent to the original trace τ. There are different flavors of progress-insensitivity in the literature. Some definitions consider program termination or divergence to be an observable event [4], [26], while other definitions, in line with ours, do not [3], [46]. They therefore ignore *pure* progress leaks, i.e., progress leaks not related to divergence/termination. All these definitions are, in any case, subject to *brute-forcing* leaks with known information-theoretic bounds [4].

C. Security condition

Our security condition ensures that changes in a user's knowledge comply with the current security policy. The condition is inspired by existing IFC conditions for dynamic policies [3], [11].

We interpret security policies with respect to initial system states. The *allowed knowledge* $A_{u,sec}$ determines the set of initial system states that a user u considers possible for a given policy sec. Given a system state $s_0 = \langle db_0, U_0, sec_0, T_0, V_0\rangle$, a security policy sec, and a user u, we define the set $A_{u,sec}(s_0)$ as $\{s \mid s \approx_{sec,u} s_0\}$, where $\langle db', U', sec', T', V'\rangle \approx_{sec,u} \langle db'', U'', sec'', T'', V''\rangle$ iff $db' \approx_{auth(sec,u)} db''$. We call $A_{u,sec}(s_0)$ *allowed* knowledge since it represents the knowledge of the initial system state that the user u is permitted to learn given the policy sec. In contrast to $[s_0]_{\approx_u}$, $A_{u,sec}(s_0)$ contains the system states that agree with s_0 with respect to the policy sec instead of the policy in s_0.

We now introduce our security condition.

Definition 2. A sequence of programs $C \in Com^{*}_{UID}$ is *secure with respect to a user u* for a scheduler $\mathcal{S}$ and a system state s_0 iff whenever $r = \langle C, M_0, \langle s_0, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau}{}^{n} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$, then for all $1 \leq i \leq n$, $K_u(\langle M_0, s_0\rangle, C, \mathcal{S}, trace(r^{i-1})) \cap A_{u,sec}(s_0) \subseteq K_u(\langle M_0, s_0\rangle, C, \mathcal{S}, trace(r^{i}))$, where the database state in r's $(i-1)$-th configuration is $\langle db, U, sec, T, V\rangle$.

Our condition ensures that a user's knowledge after observing $trace(r^i)$ is no more precise than his previous knowledge combined with the allowed knowledge from r's $(i-1)$-th configuration, i.e., the knowledge increase is allowed by the current policy.

V. Enforcement

We now present a monitor that provably secures WhileSQL programs. To achieve end-to-end security across the database and applications, our monitor tracks dependencies at the database level (between tuples and queries) and at the program level (between variables). It ensures that the information released by output statements and public events complies with the current security policy.

The monitor instruments WhileSQL programs to track dependencies between variables, and it blocks the execution of statements that may leak sensitive information. The monitor

also intercepts each database command and expands it into WHILESQL code to prevent leaks caused by triggers and other database side-effects. While executing the code produced during expansion, the monitor tracks the dependencies between variables and queries.

This approach cleanly separates the application's code and the security policy, thus putting trust in the security monitor instead of the application. This trust is formally justified by proving that the security monitor satisfies our security condition. Our monitor also supports a rich class of policies, including dynamic policy changes. The policies are expressed using `GRANT` and `REVOKE` commands, and the monitor ensures their end-to-end interpretation through the application-database boundary. This approach is transparent to the applications and does not require customized database support.

A. Preliminaries

We leverage *disclosure lattices* to reason about the information disclosed by sets of queries [8]. Recall that a security policy specifies a set of database tables and views that a user is authorized to read. Hence, policies can be seen as sets of database queries, which are elements of a disclosure lattice. This natural connection between disclosure lattices, queries, and policies allows us to track cumulative information disclosures across multiple queries and determine whether a new query would increase the total amount of information beyond what is actually allowed by the policy. Additionally, disclosure lattices allow us to track fine-grained dependencies across the application and the database. This is needed to enforce realistic security policies, such as row-level database policies. We discuss the benefits of using disclosure lattices for IFC in §V-C. In the following, we fix a database configuration $\langle D, \Gamma \rangle$ and we refer only to database states db defined over the schema D and that satisfy the integrity constraints in Γ.

Predicate queries. A *predicate query* is a query of the form $T(\overline{v})$, where T is a table identifier in D and $\overline{v} \in Val^{|T|}$ is a tuple of values whose length is T's arity $|T|$. A predicate query represents a single tuple in the database. The set of all predicate queries is RC^{pred}.

Determinacy. Query determinacy [35] is the task of determining, given two sets of queries Q and Q', if the results of the queries in Q are always *sufficient* to determine the result of the queries in Q'. Formally, Q determines Q', written $Q \twoheadrightarrow Q'$, iff for all database states db, db', if $[q]^{db} = [q]^{db'}$ for all $q \in Q$, then $[q']^{db} = [q']^{db'}$ for all $q' \in Q'$, where $[q]^{db}$ denotes q's result in db. For instance, the set $\{T(1), R(2)\}$ determines the query $T(1) \vee R(2)$. In general, determinacy is different from logical entailment, e.g., $T(1) \models T(1) \vee R(2)$ but $T(1) \not\twoheadrightarrow T(1) \vee R(2)$.

Query support. The support of a query q contains all tuples that may influence q's results. To precisely capture a query's support, we first introduce the notion of minimal determinacy. A set of predicate queries Q *minimally determines* q, denoted $minDet(Q, q)$, iff Q is the smallest set that determines q. Formally, $minDet(Q, q)$ iff $Q \twoheadrightarrow q$ and there is no $Q' \subset Q$ such that $Q' \twoheadrightarrow q$.The *support of* q, denoted $supp(q)$, contains all sets of tuples that minimally determine q, i.e., $supp(q) := \{Q \in 2^{RC^{pred}} \mid minDet(Q, q)\}$.That is, $supp(q)$ contains all and only those tuples that may influence q's outcome. For instance, the query $T(1) \vee R(2)$ is minimally determined by $\{T(1), R(2)\}$. Hence, its support is $\{\{T(1), R(2)\}\}$.

We consider only sets of integrity constraints Γ such that $supp(q) = \{\{q\}\}$ for all predicate queries $q \in RC^{pred}$. Integrity constraints commonly used in practice, such as primary and foreign keys, satisfy this requirement. This guarantees that the information associated with a predicate query depends just on the query itself.

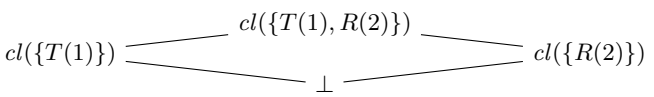

Fig. 3: Disclosure lattice for the queries $T(1)$ and $R(2)$.

Disclosure orders and lattices. Bender et al. [8] recently introduced disclosure orders and lattices to reason about the information disclosed by queries. Given two sets of queries Q_1 and Q_2, disclosure lattices provide a precise model for answering questions such as "Does Q_1 reveal more information than Q_2?" or "What is the combined and the common information that is disclosed by both Q_1 and Q_2?"

A *disclosure order* [8] is a binary relation $\preceq$ over sets of queries (i.e., over 2^{RC} where RC is the set of all queries), such that: (1) for all $Q, Q' \in 2^{RC}$, if $Q \subseteq Q'$, then $Q \preceq Q'$, (2) for all $Q, Q', Q'' \in 2^{RC}$, if $Q \preceq Q'$ and $Q' \preceq Q''$, then $Q \preceq Q''$, and (3) for all $Q, Q', Q'' \in 2^{RC}$, if $Q \preceq Q''$ and $Q' \preceq Q''$, then $Q \cup Q' \preceq Q''$.

A disclosure order $\preceq$ is, in general, not anti-symmetric. Hence, as is standard in lattice theory [18], we introduce the concept of closure, which we use to construct a lattice. Given a set of queries Q and a disclosure order $\preceq$, the *closure* of Q, written $cl(Q)$, is $\{q \in RC \mid \{q\} \preceq Q\}$. The $\preceq$*-disclosure lattice* [8] is a tuple $\langle \mathcal{L}, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ where (1) $\mathcal{L} = \{cl(Q) \mid Q \in 2^{RC}\}$, (2) $cl(Q) \sqsubseteq cl(Q')$ iff $Q \preceq Q'$, (3) $cl(Q) \sqcap cl(Q') = cl(Q) \cap cl(Q')$, (4) $cl(Q) \sqcup cl(Q') = cl(Q \cup Q')$, (5) $\bot = cl(\emptyset)$, and (6) $\top = cl(RC)$.

Determinacy induces an ordering on the information content of queries. Hence, it is a good candidate for defining disclosure lattices. Formally, we define the determinacy-based disclosure order using the relation $\preceq^{\twoheadrightarrow}$: given $Q, Q' \in 2^{RC}$, $Q \preceq^{\twoheadrightarrow} Q'$ iff $Q' \twoheadrightarrow Q$. Note that $Q \preceq^{\twoheadrightarrow} Q'$ means that Q is less informative than Q'. As shown in [8], $\preceq^{\twoheadrightarrow}$ is a disclosure order and the corresponding disclosure lattice is complete. Figure 3 depicts the portion of the lattice involving the queries $T(1)$ and $R(2)$.

B. Security monitor

We now present our dynamic security monitor. For simplicity, we consider a single attacker, denoted by the user atk. We denote by sec_0 the initial security policy.

Security lattice. Our security monitor uses the disclosure lattice to track information. As a security lattice, we use the

disclosure lattice $\langle \mathcal{L}, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ defined over the database schema D, where $\sqsubseteq$ is $\preceq^{\rightarrow}$. Since query determinacy is undecidable in general [35], in §VI we present a practical approximation for handling disclosure lattices.

Monitor states. A *monitor state* Δ is a function $Var \cup RC^{pred} \cup \{\mathtt{pc}_u \mid u \in UID\} \to \mathcal{L}$ that associates each variable and predicate query (which represents a tuple) with a label. The monitor state also stores the label associated with the security context of each program. Since each user u executes only one program, we formalize the program's security context using identifiers of the form $\mathtt{pc}_u$, where $u \in UID$ is the user executing the program. For example, $\Delta(\mathtt{pc}_{\mathtt{Bob}})$ captures the label associated with the condition of an **if** statement if Bob's program is executing a branch of the **if** statement. We lift Δ to expressions: $\Delta(e) = \bigsqcup_{x \in vars(e)} \Delta(x)$, where e is an expression and $vars(e)$ are its free variables. The monitor's initial state Δ_0 is as follows: (a) for each $x \in Var$, $\Delta_0(x) = \bot$, (b) for all $q \in RC^{pred}$, $\Delta_0(q) = cl(q)$, and (c) for all $u \in UID$, $\Delta_0(\mathtt{pc}_u) = \bot$.

Mapping queries to labels. Our security monitor tracks only dependencies between predicate queries, i.e., tuples. Hence, we use the function $L_{\mathcal{Q}}$ to derive the label associated with general queries: $L_{\mathcal{Q}}(\Delta, q) = \bigsqcup_{Q \in supp(q)} \bigsqcup_{q' \in Q} \Delta(q')$. The function associates to a query q the join of the labels associated with all predicate queries in q's support. This ensures that $L_{\mathcal{Q}}(\Delta, q)$ accounts for the labels of all predicate queries that may influence q's results. For instance, given a monitor state Δ, the query $T(1) \vee R(2)$, whose support is $\{\{T(1), R(2)\}\}$, is associated with the label $\Delta(T(1)) \sqcup \Delta(R(2))$, thus capturing that it reveals information about $T(1)$ and $R(2)$. For predicate queries $T(\overline{v})$, $L_{\mathcal{Q}}(\Delta, T(\overline{v})) = \Delta(T(\overline{v}))$.

Mapping users to labels. The function $L_{\mathcal{U}}$ maps users to labels in our security lattice. Since we are interested in end-to-end security guarantees, we associate to the attacker atk the set of tables and views he is authorized to read according to the current access control policy and to the initial policy sec_0. Formally, $L_{\mathcal{U}}(s, u) = \top$ for any $u \notin \{atk, public\}$. For the attacker atk, $L_{\mathcal{U}}(s, atk) = cl(auth(s, atk) \cup auth(sec_0, atk))$, which captures what the attacker can observe according to the initial policy sec_0 and the policy in s. Finally, $L_{\mathcal{U}}(s, public) = L_{\mathcal{U}}(s, atk)$. For example, given a security policy sec_0 stating that the attacker atk can read the table T but not the table R, $L^{sec_0}_{\mathcal{U}}(s, atk) = \bigsqcup_{v \in Val} cl(T(v))$. In the following, we omit the reference to sec_0 when this is clear from the context, i.e., we write $L_{\mathcal{U}}(s, u)$ instead of $L^{sec_0}_{\mathcal{U}}(s, u)$.

The mappings $L_{\mathcal{Q}}$ and $L_{\mathcal{U}}$ allow us to reason about information disclosure. For instance, if the above attacker observes the result of the query $q = \mathtt{SELECT}\ T(1) \vee R(2)$ when the monitor state is Δ_0, this violates the security policy. In fact, $L_{\mathcal{Q}}(\Delta_0, q) \not\sqsubseteq L_{\mathcal{U}}(s, atk)$, since $cl(\{T(1), R(2)\}) \not\sqsubseteq \bigsqcup_{v \in Val} cl(T(v))$.

Expansion process. To correctly handle triggers, our monitor rewrites each SQL command into WHILESQL statements encoding the triggers' execution. We do so using the $expand(s, m, u, x \leftarrow q)$ function, which takes as input a system state s, a memory m, a user u, and a statement $x \leftarrow q$, and produces as output the statements modeling the triggers' execution and database's other side effects.

In a nutshell, the *expand* function works as follows. First, depending on the query q and the database configuration in s, *expand* computes all possible *execution paths*, which are sequences of queries and triggers together with their results. In particular, a query may successfully execute or generate an integrity or a security exception. Triggers additionally may not be enabled, that is they are not executed since their condition is not satisfied. Afterward, *expand* translates each execution path into an **if** statement. For each execution path, the **if**'s body contains the WHILESQL statements implementing the execution of the queries and the triggers as described in the path. In contrast, the **if**'s condition checks whether the weakest precondition for the actual execution of the path is met. For instance, the code checks whether the condition of an enabled trigger is actually satisfied or whether executing a command would lead to an integrity exception if the execution path says so. To achieve this, we designed a procedure for computing the weakest precondition starting from execution paths. This can always be automatically computed since execution paths are loop-free. We formalize $expand(s, m, u, x \leftarrow q)$ and prove its correctness in [23]. Example 1 concretely illustrates how *expand* works.

Additional queries and statements. Our monitor extends WHILESQL with two designated queries $T \oplus \overline{e}$ and $T \ominus \overline{e}$, and four designated statements **asuser**(u', c), $\|x \leftarrow q\|$, $[c]$, and **set pc to** l. The $T \oplus \overline{e}$ (respectively $T \ominus \overline{e}$) query inserts into (respectively deletes from) the table T the tuple $\overline{e}$ without database-level side effects like firing triggers or throwing exceptions in case integrity constraints are violated. The **asuser**(u', c) statement is used to execute the command c as the user u' (inside the session of the user u executing the **asuser**(u', c) statement). Finally, the $\|x \leftarrow q\|$ statement, where x is a variable and q is a query, denotes a query statement that has already been processed by *expand*. All the above queries and statements are used during the expansion process.

To avoid *internal* timing leaks caused by executing multiple programs in parallel [39], the monitor's semantics executes branching statements atomically, i.e., without interleaving the execution of other programs whenever a program is executing a branching statement. To do so, we introduce statements of the form $[c]$ denoting that the command c should be executed atomically, and statements **set pc to** l, where l is a label in $\mathcal{L}$, which are used to update the label associated to the program's context.

Enforcement rules. Figure 4 presents selected rules from our monitor's semantics. The rules use the auxiliary functions $L_{\mathcal{U}}$ and $L_{\mathcal{Q}}$ to derive the security labels associated with users and queries. We present the full operational semantics in [23].

The rule F-ASSIGN updates the monitor's state whenever there is an assignment. This rule prevents leaks using No-Sensitive Upgrade (NSU) checks [50]. The rule F-OUT ensures that the monitor produces only secure output events. It outputs the value of the expression e to the user u' only if the security labels associated with e and the program counter

F-ASSIGN
$$\frac{\Delta(\mathtt{pc}_u) \sqsubseteq \Delta(x) \qquad \Delta' = \Delta[x \mapsto \Delta(\mathtt{pc}_u) \sqcup \Delta(e)]}{\langle \Delta, x := e, m, s\rangle \rightsquigarrow_u \langle \Delta', \varepsilon, m[x \mapsto [\![e]\!](m)], s\rangle}$$

F-OUT
$$\frac{\Delta(e) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, u')}{\langle \Delta, \mathbf{out}(u', e), m, s\rangle \overset{\langle u', [\![e]\!](m)\rangle}{\rightsquigarrow_u} \langle \Delta, \varepsilon, m, s\rangle}$$

F-EXPAND
$$\frac{c_e = expand(s, x, q, u)}{\langle \Delta, x \leftarrow q, m, s\rangle \rightsquigarrow_u \langle \Delta, [c_e], m, s\rangle}$$

F-IFTRUE
$$\frac{[\![e]\!](m) = \mathbf{tt} \qquad c' = [c_1 \ ; \mathbf{set\ pc\ to}\ \Delta(\mathtt{pc}_u)] \qquad \Delta' = \Delta[\mathtt{pc}_u \mapsto \Delta(e) \sqcup \Delta(\mathtt{pc}_u)]}{\langle \Delta, \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, m, s\rangle \rightsquigarrow_u \langle \Delta', c', m, s\rangle}$$

F-SELECT
$$\frac{\begin{array}{c}\{v_1, \ldots, v_n\} = vars(\varphi) \\ \varphi' = \varphi[v_1 \mapsto [\![v_1]\!](m), \ldots, v_n \mapsto [\![v_n]\!](m)] \qquad q = \mathtt{SELECT}\ \varphi \qquad [\![q]\!](s, u) = \langle s', r, \epsilon\rangle \qquad \ell_\varphi = L_{\mathcal{Q}}(\Delta, \varphi) \sqcup \bigsqcup_{v \in vars(\varphi)} \Delta(v) \qquad \Delta(\mathtt{pc}_u) \sqsubseteq \Delta(x)\end{array}}{\langle \Delta, \|x \leftarrow \mathtt{SELECT}\ \varphi\|, m, s\rangle \rightsquigarrow_u \langle \Delta[x \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_\varphi], \varepsilon, m[x \mapsto r], s'\rangle}$$

F-UPDATEDATABASEOK
$$\frac{\begin{array}{c}\overline{v} = \langle [\![e_1]\!](m), \ldots, [\![e_n]\!](m)\rangle \\ \otimes \in \{\oplus, \ominus\} \qquad [\![T \otimes \overline{v}]\!](s, u) = \langle s', r, \epsilon\rangle \qquad \ell_e = \bigsqcup_{1 \le i \le n} \Delta(e_i) \qquad \ell_e \sqsubseteq \Delta(T(\overline{v})) \qquad \Delta(\mathtt{pc}_u) \sqsubseteq \Delta(T(\overline{v})) \qquad \Delta(\mathtt{pc}_u) \sqsubseteq \Delta(x)\end{array}}{\langle \Delta, \|x \leftarrow T \otimes \langle e_1, \ldots, e_n\rangle\|, m, s\rangle \rightsquigarrow_u \langle \Delta[T(\overline{v}) \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_e, x \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_e], \varepsilon, m[x \mapsto r], s'\rangle}$$

F-UPDATECONFIGURATIONOK
$$\frac{\begin{array}{c}\{v_1, \ldots, v_n\} = vars(q) \qquad q' = q[v_1 \mapsto [\![v_1]\!](m), \ldots, v_n \mapsto [\![v_n]\!](m)] \qquad isCfgCmd(q') \\ [\![q']\!](s, u) = \langle s', r, \epsilon\rangle \qquad \ell_{cmd} = \bigsqcup_{1 \le i \le n} \Delta(v_i) \qquad \ell_{cmd} \sqsubseteq cl(auth(sec_0, atk)) \qquad \Delta(\mathtt{pc}_u) \sqsubseteq cl(auth(sec_0, atk)) \qquad \Delta(\mathtt{pc}_u) \sqsubseteq \Delta(x)\end{array}}{\langle \Delta, \|x \leftarrow q\|, m, s\rangle \overset{\langle public, q'\rangle}{\rightsquigarrow_u} \langle \Delta[x \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_{cmd}], \varepsilon, m[x \mapsto r], s'\rangle}$$

Fig. 4: Security monitor – selected rules.

are authorized to flow to u', i.e., $\Delta(e) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, u')$. The rule F-IFTRUE, instead, executes the **then** branch c_1 in an **if** statement and updates the labels of $\mathtt{pc}_u$ based on the label of the **if**'s condition. The rule relies on the **set pc to** l command to reset the label of $\mathtt{pc}_u$ when leaving the **then** branch. Note that the rule encapsulates both the **then** branch c_1 and the **set pc to** l statement inside an atomic statement $[c_1$; **set pc to** $l]$ to prevent internal timing channels caused by the scheduler. We remark that the above rules implement standard dynamic information-flow tracking [38].

The rule F-EXPAND ensures that triggers as well as integrity constraint checking is de-sugared into WHILESQL code using the *expand* function. The F-SELECT rule ensures, using NSU checks, that the queries' results are stored only in variables with the proper security labels. The rule, finally, updates the label of the variable storing the query's result to correctly propagate the flow of information.

The rule F-UPDATECONFIGURATIONOK handles configuration commands, i.e., `GRANT`, `REVOKE`, and `CREATE` commands. Since configuration changes are visible to atk (i.e., the rule produces a public observation), the rule ensures that such changes are performed only in contexts that are initially low for the attacker, i.e., $\Delta(\mathtt{pc}_u) \sqsubseteq cl(auth(sec_0, atk))$. Furthermore, the rule prevents leaks of sensitive information using the free variables in the commands by checking that $\ell_{cmd} \sqsubseteq cl(auth(sec_0, atk))$. The rule also uses NSU checks to ensure that the query's results are stored only in variables with the proper security labels. The rule uses the predicate $isCfgCmd(q)$, which returns $\top$ iff q is a configuration command. Finally, the rule F-UPDATEDATABASEOK handles queries that modify the database content. The rule ensures that there are no changes to the security labels based on secret information using NSU checks. The rule keeps also track of the labels associated with the information stored in the database by updating the monitor's state Δ.

In WHILESQL, policy changes are publicly visible. This eliminates leaks through authorization channels [2], and no additional checks (cf. *channel context bounds* [3]) are needed.

Theorem 1, proven in [23], states that our monitor is sound: it satisfies Def. 2 with $\rightsquigarrow$ as the evaluation relation.

Theorem 1. *For all sequences of programs $C \in Com^*_{UID}$, schedulers $\mathcal{S}$, sequences of memories $M \in Mem^*_{UID}$, and system states s, whenever $r = \langle \Delta_0, C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \overset{\tau}{\rightsquigarrow}^n \langle \Delta', C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$, then for all $1 \le i \le n$, $K^{\rightsquigarrow}_{atk}(\langle M, s\rangle, C, \mathcal{S}, trace(r^{i-1})) \cap A_{atk,sec}(s) \subseteq K^{\rightsquigarrow}_{atk}(\langle M, s\rangle, C, \mathcal{S}, trace(r^i))$, where $K^{\rightsquigarrow}_{atk}$ refers to Def. 1 with $\rightsquigarrow$ as evaluation relation and the system state in r's $(i-1)$-th configuration is $\langle db, U, sec, T, V\rangle$.*

Example 1. Let T, V, Z be three tables, t be the trigger defined by the administrator using the command `CREATE TRIGGER` t `ON` T `AFTER INSERT IF` $V(1)$ `DO` $\{$`INSERT 1 INTO` $Z\}$, and s be a state containing t. In this context, the statement $x \leftarrow$ `INSERT 2 INTO` T is expanded as follows (provided that all commands are authorized by the policy and there are no integrity constraints): $\|y \leftarrow$ `SELECT` $V(1)\|$; **if** y **then** $\{\|x \leftarrow T \oplus 2\|$; **asuser**$(admin, \|z \leftarrow Z \oplus 1\|)\}$ **else** $\{\|x \leftarrow T \oplus 2\|\}$.

Suppose the attacker atk executes $x \leftarrow$ `INSERT 2 INTO` $T; w \leftarrow$ `SELECT` $Z(1)$; **out**(atk, w) from a system state s_0 where the tables T and Z are empty and the table V contains a single record with value 1. We illustrate the

monitor's behavior for the security policy where atk cannot read V but can read and modify T and Z. In this case, the program is insecure since the presence of 1 in Z depends (implicitly) on the presence of 1 in V, which atk cannot read.

Consider the program execution with the initial state s_0 as above, and the initial monitor state Δ_0 such that $\Delta_0(\mathtt{pc}_{atk}) = \bot$. The attacker's label is $L_{\mathcal{U}}(s_0, atk) = \bigsqcup_{v \in Val} cl(T(v)) \sqcup \bigsqcup_{v \in Val} cl(Z(v))$. The monitor would apply the rules F-EXPAND (explained above), F-SELECT, F-IFTRUE, F-UPDATEDATABASEOK, F-ASUSER (not shown), F-UPDATE-DATABASEOK, F-SETPC (not shown), F-SELECT, and F-OUT. The evaluation of the first `SELECT` statement yields $\Delta' = \Delta_0[y \mapsto \Delta(V(1)) \sqcup \bot]$, i.e., $\Delta'(y) = cl(V(1))$. The evaluation of the boolean condition y yields $\Delta' = \Delta[y \mapsto cl(V(1)), \mathtt{pc}_{atk} \mapsto cl(V(1))]$. For the subsequent database update, the monitor checks whether $\Delta'(\mathtt{pc}_{atk}) \sqsubseteq \Delta'(T(2))$, namely, whether $cl(V(1)) \sqsubseteq cl(T(2))$. Since this is not the case, the monitor stops the execution and prevents the leakage. ■

C. Discussion

Supported policies. Our monitor supports dynamic policies expressed using `GRANT` and `REVOKE` commands. It also supports row-level policies, which can be expressed using views that disclose a subset of the tuples in a table.

Our monitor associates security labels with tuples. It does not label columns and therefore it cannot enforce column-level policies, which disclose only selected attributes of a table, in their full generality. Despite that, many column-level policies can be translated into equivalent row-level policies by carefully refactoring the database schema. We illustrate this with an example. Consider a table $\mathtt{PERSON}(\mathtt{id}, \mathtt{name}, \mathtt{salary})$, with primary key `id`, where the attributes `id` and `name` are public, while the attribute `salary` is secret. We can refactor the table `PERSON` into two tables $\mathtt{PERSON}_{\mathtt{public}}(\mathtt{id}, \mathtt{name})$ and $\mathtt{PERSON}_{\mathtt{secret}}(\mathtt{id}, \mathtt{salary})$. Then, the column-level policy can be enforced using row-level policies by granting access only to $\mathtt{PERSON}_{\mathtt{public}}$ and not to $\mathtt{PERSON}_{\mathtt{secret}}$. More generally, column-level policies can be encoded as row-level policies (and enforced by our monitor) whenever the table's primary key is public, and the column-level policy does not change during the execution.

Disclosure lattices. Disclosure lattices allow us to express fine-grained tuple-level dependencies between data and variables, such as "the value of the variable x may depend on the initial values of the queries $T(1)$ and $V(2)$, but not on the value of the query $R(3)$." Our monitor leverages disclosure lattices to record all the data that may have influenced a variable's current value. In contrast, existing approaches, such as [7], [43], track column-level dependencies using the standard "low" and "high" labels.

While these two approaches are incomparable precision-wise (see [23]), by tracking tuple-level dependencies, we can directly support row-level policies, which are a common policy idiom from database security, and form the basis of many fine-grained database access control models [12], [24], [37], [48]. Row-level policies cannot be easily supported using column-level dependency tracking since there is no way to assign distinct security labels to subsets of tuples in a table. Additionally, we can also enforce static column-level policies by refactoring the database schema.

Multiple attackers. To ease the presentation, our monitor considers a fixed attacker atk. Specifically, Theorem 1 guarantees that atk cannot access sensitive information and that other users' programs do not reveal sensitive information to atk.

To handle arbitrary attackers, we can replace all checks of the form $\ell \sqsubseteq cl(auth(sec_0, atk))$ with $\bigwedge_{u \in U} \ell \sqsubseteq cl(auth(sec_0, u))$, all checks of the form $\ell \sqsubseteq L_{\mathcal{U}}(s, public)$ with $\bigwedge_{u \in U} \ell \sqsubseteq cl(auth(sec_0, u) \cup auth(sec, u))$, and all checks of the form $\ell \sqsubseteq L_{\mathcal{U}}(s, u)$, where $u \neq public$, with $\ell \sqsubseteq cl(auth(sec_0, u) \cup auth(sec, u))$, where U is the set of users, sec_0 is the initial policy, sec is the policy in the state s. This guarantees that each user accesses only the information he is authorized to access by the policy, i.e., it ensures that our security condition is satisfied for all users u.

VI. DISCLOSURE LATTICES IN PRACTICE

Our monitor tracks fine-grained dependencies between tuples and variables using disclosure lattices. However, directly computing with disclosure lattices is challenging. For instance, checking $l_1 \sqsubseteq l_2$ and computing $L_{\mathcal{Q}}(\Delta, q)$ both requires solving query determinacy, which is undecidable in general. We now propose a practical way of approximating computations over disclosure lattices.

A. Approximating disclosure lattices

Our security monitor in §V relies on disclosure lattices for several purposes. The monitor state Δ maps variables and tuples to labels in the lattice $\mathcal{L}$. Additionally, security checks are implemented using the lattice's ordering relation $\sqsubseteq$, and label updates are implemented using the lattice's join operator $\sqcup$. Finally, we map queries and users to labels using the $L_{\mathcal{Q}}$, $L_{\mathcal{U}}$, and $auth$ functions.

An approximation of the (determinacy-based) disclosure lattice provides lower and upper bounds for each of the aforementioned components. Formally, an *approximation* is a tuple $\langle \mathcal{L}^{abs}, \sqsubseteq^{abs}, \sqcup^{abs}, \Delta_0^{abs}, L_{\mathcal{Q}}^{abs}, L_{\mathcal{U}}^{abs}, auth^{abs}, \gamma^-, \gamma^+ \rangle$, where $\mathcal{L}^{abs}$ is the set of abstract labels, $\sqsubseteq^{abs}$ is a preorder over abstract labels, $\sqcup^{abs}$ is the join operator over abstract labels, $L_{\mathcal{Q}}^{abs}$ maps abstract monitor states and queries to abstract labels, $L_{\mathcal{U}}^{abs}$ maps system states and users to abstract labels, and $auth^{abs}$ maps policies and users to abstract labels. Finally, $\gamma^- : \mathcal{L}^{abs} \to \mathcal{L}$ and $\gamma^+ : \mathcal{L}^{abs} \to \mathcal{L}$ provide respectively lower and upper bounds on the information content of abstract labels in terms of the disclosure lattice $\mathcal{L}$. An abstract label $\ell \in \mathcal{L}^{abs}$ represents all concrete labels $l \in \mathcal{L}$ such that $\gamma^-(\ell) \sqsubseteq l \sqsubseteq \gamma^+(\ell)$.

We remark that we need both under- and over-approximations to soundly check containment between labels since abstract labels may occur on both sides of $\sqsubseteq^{abs}$.

B. Symbolic tuples

Symbolic tuples. Our approximation relies on symbolic tuples, which concisely represent sets of concrete tuples (i.e., predicate

queries) using logical formulae. Formally, a *symbolic tuple* is a pair $\langle T, \varphi \rangle$, where T is a table identifier of arity n and φ is a boolean combination of equality and inequality constraints over variables in $\{x_1, \ldots, x_n\}$ and values in Val. We denote by ST_D the set of all symbolic tuples defined over the database schema D.

The *concretization of a symbolic tuple* $\langle T, \varphi \rangle$, denoted $\gamma(\langle T, \varphi \rangle)$, is the set $\{T(v_1, \ldots, v_{|T|}) \mid v_1, \ldots, v_{|T|} \in Val \wedge \models \varphi[x_1 \mapsto v_1, \ldots, x_{|T|} \mapsto v_{|T|}]\}$ containing all possible concrete tuples that satisfy the constraint φ, where $\models \varphi$ denotes that φ is a valid formula.

For instance, the symbolic tuple $\langle T, x_1 \neq x_2 \rangle$ represents the set of all concrete tuples $T(v_1, v_2)$ such that $v_1 \neq v_2$. The concrete tuple $T(1, 2)$ belongs to $\langle T, x_1 \neq x_2 \rangle$'s concretization $\gamma(\langle T, x_1 \neq x_2 \rangle)$, while the concrete tuple $T(1, 1)$ does not.

Abstract labels. In our approximation, we track lower and upper bounds using two sets of symbolic tuples. Formally, a label ℓ is a pair $\langle S^-, S^+ \rangle$ such that S^- and S^+ are sets of symbolic tuples. The set S^- captures ℓ's lower bounds whereas S^+ captures ℓ's upper bounds. Given a label $\ell = \langle S^-, S^+ \rangle$, we denote by $\ell|_-$ (respectively $\ell|_+$) the set S^- (respectively S^+).

We can now formalize the lower and upper bound concretization functions γ^- and γ^+. For an abstract label ℓ, its lower-bound (respectively upper-bound) concretization $\gamma^-(\ell)$ (respectively $\gamma^+(\ell)$) is the (closure of the) union of the concretizations of all tuples in $\ell|_-$ (respectively $\ell|_+$). That is, $\gamma^-(\ell) = cl(\bigcup_{\langle T,\varphi\rangle \in \ell|_-} \gamma(\langle T, \varphi \rangle))$ and $\gamma^+(\ell) = cl(\bigcup_{\langle T,\varphi\rangle \in \ell|_+} \gamma(\langle T, \varphi \rangle))$.

The set $\mathcal{L}^{abs}$ of all valid abstract labels contains all labels ℓ for which the lower-bound concretization is below its upper-bound concretization with respect to the concrete ordering $\sqsubseteq$, i.e., $\mathcal{L}^{abs} := \{\langle S^-, S^+ \rangle \in \mathcal{P}(ST_D)^2 \mid \gamma^-(\langle S^-, S^+ \rangle) \sqsubseteq \gamma^+(\langle S^-, S^+ \rangle)\}$.

Consider the abstract label $\ell = \langle \{\langle T, x_1 = 2 \rangle\}, \{\langle T, \top \rangle, \langle R, \top \rangle\} \rangle$. It represents all concrete labels l such that $cl(\{T(2, x_2) \mid x_2 \in Val\}) \sqsubseteq l \sqsubseteq cl(\{T(x_1, x_2) \mid x_1, x_2 \in Val\}) \sqcup cl(\{R(x_1) \mid x_1 \in Val\})$. This implies, for instance, that ℓ at most contains as much information as the tables T and R. However $\ell'' = \langle \{\langle T, \top \rangle, \langle R, \top \rangle\}, \{\langle T, x_1 = 2 \rangle\} \rangle$ is not a valid abstract label since $\gamma^-(\ell'') \not\sqsubseteq \gamma^+(\ell'')$.

Ordering relation. The abstract ordering relation $\sqsubseteq^{abs}$ is as follows: $\langle S_1^-, S_1^+ \rangle \sqsubseteq^{abs} \langle S_2^-, S_2^+ \rangle$ iff for all symbolic tuples $\langle T, \varphi \rangle \in S_1^+$, there is a symbolic tuple $\langle T, \varphi' \rangle \in S_2^-$ such that $\varphi \models \varphi'$, where $\varphi \models \varphi'$ denotes that any assignment that satisfies φ also satisfies φ' (this is equivalent to $\gamma(\langle T, \varphi \rangle) \subseteq \gamma(\langle T, \varphi' \rangle)$). This ensures that whenever $\langle S_1^-, S_1^+ \rangle \sqsubseteq^{abs} \langle S_2^-, S_2^+ \rangle$ is satisfied, then $\gamma^+(\langle S_1^-, S_1^+ \rangle) \sqsubseteq \gamma^-(\langle S_2^-, S_2^+ \rangle)$ holds as well. Hence, the concrete tuples represented by $\langle S_1^-, S_1^+ \rangle$ are below those represented by $\langle S_2^-, S_2^+ \rangle$.

To illustrate, consider the abstract labels $\ell_1 = \langle \emptyset, \{\langle T, x_1 = 2 \wedge x_2 \neq x_1 \rangle\} \rangle$ and $\ell_2 = \langle \{\langle T, x_1 = 2 \rangle\}, \{\langle T, \top \rangle, \langle R, \top \rangle\} \rangle$. It is easy to see that $\ell_1 \sqsubseteq^{abs} \ell_2$ holds: any concrete tuple in $\gamma^+(\ell_1)$ also belongs to $\gamma^-(\ell_2)$ since any satisfying assignment for $x_1 = 2 \wedge x_2 \neq x_1$ also satisfies $x_1 = 2$. In contrast, $\ell_2 \not\sqsubseteq^{abs} \ell_1$. For instance, $T(1, 1)$ belongs to $\gamma^+(\ell_2)$ but not to $\gamma^-(\ell_1)$.

Join operator. The join operator between abstract labels is the pairwise union of their components: given two labels $\ell_1 = \langle S_1^-, S_1^+ \rangle, \ell_2 = \langle S_2^-, S_2^+ \rangle \in \mathcal{L}^{abs}$, their join $\ell_1 \sqcup^{abs} \ell_2$ is $\langle S_1^- \cup S_2^-, S_1^+ \cup S_2^+ \rangle$. For instance, given two abstract labels $\ell_1 = \langle \{\langle T, x_1 = 2 \rangle\}, \{\langle T, \top \rangle\} \rangle$ and $\ell_2 = \langle \{\langle T, x_1 \neq x_2 \rangle\}, \{\langle T, \top \rangle\} \rangle$, the label $\ell_1 \sqcup^{abs} \ell_2$ is $\langle \{\langle T, x_1 = 2 \rangle, \langle T, x_1 \neq x_2 \rangle\}, \{\langle T, \top \rangle\} \rangle$.

Labeling queries. To map queries to labels, we need both lower and upper bounds for $L_\mathcal{Q}$. In the following, let Δ^{abs} be an abstract monitor state and q be a boolean query. Moreover, we denote $L_\mathcal{Q}$'s lower and upper bounds respectively by $\ell^-_{\Delta^{abs},q}$ and $\ell^+_{\Delta^{abs},q}$. Namely, $L^{abs}_\mathcal{Q}(\Delta^{abs}, q) = \langle \ell^-_{\Delta^{abs},q}, \ell^+_{\Delta^{abs},q} \rangle$. We formalize $\ell^-_{\Delta^{abs},q}$ and $\ell^+_{\Delta^{abs},q}$ below. Without loss of generality, we assume that universally quantified statements $\forall \overline{x}.\ \varphi$ are expressed as $\neg\exists\overline{x}.\neg\varphi$.

Over-approximating queries. We compute the upper bound of $L_\mathcal{Q}$ in two steps. We first extract the symbolic tuples from the query q. We then compute $\ell^+_{\Delta^{abs},q}$ by accounting for the labels in Δ^{abs}.

Given a query q, the function $cstrs(q)$ extracts the symbolic tuples from q. We denote by $subf(q)$ the set of q's immediate sub-formulae that contain predicate symbols. Moreover, $nf(q)$ denotes that q is of the form $T(\overline{x}) \wedge \varphi$, where φ is a (possibly empty) boolean combination of equalities and inequalities over variables in $\overline{x}$ and values in Val. The set $cstrs(q)$ is recursively defined as $cstrs(q) = \left(\bigcup_{q' \in subf(q) \wedge \neg nf(q)} cstrs(q')\right) \cup \{\langle T, \varphi \rangle \mid nf(q) \wedge q = (T(\overline{x}) \wedge \varphi)\}$. Observe that the concrete tuples represented by the symbolic tuples in $cstrs(q)$ contain those in q's support. That is, $supp(q) \subseteq \bigcup_{\langle T,\varphi\rangle \in cstrs(q)} \gamma(\langle T, \varphi \rangle)$.

Given a symbolic tuple $\langle T, \varphi \rangle$ and a finite set M of predicate queries of the form $T(\overline{v})$, we denote by $R(\langle T, \varphi \rangle, M)$ the most precise symbolic tuple $\langle T, \varphi' \rangle$ such that $(\gamma(\langle T, \varphi \rangle) \setminus M) \subseteq \gamma(\langle T, \varphi' \rangle)$.

Given an abstract state Δ^{abs} and a query q, we compute $\ell^+_{\Delta^{abs},q}$ as:

$$\bigcup_{\langle T,\varphi\rangle \in cstrs(q')} \left(\bigcup_{T(\overline{v}) \in \gamma(\langle T,\varphi\rangle) \cap M_T} \Delta^{abs}(T(\overline{v}))|_+ \cup \{R(\langle T, \varphi \rangle, M_T)\} \right)$$

where q' is the query obtained by recursively replacing views with their definitions and M_T is the set $\{T(\overline{v}) \in RC^{pred} \mid \Delta^{abs}(T(\overline{v}))|_+ \neq \Delta^{abs}_0(T(\overline{v}))|_+\}$ of all predicate queries whose upper bound is different from the initial one.

To illustrate, consider the query q defined as $\exists x.(T(2, x) \wedge (x = 3 \vee x = 4)) \wedge \forall x.\ R(x) \rightarrow \exists y.\ S(3, y)$. Computing $cstrs(q)$ produces the symbolic tuples: $\{\langle T, x_1 = 2 \wedge (x_2 = 3 \vee x_2 = 4) \rangle, \langle R, \top \rangle, \langle S, x_1 = 3 \rangle\}$. Given a monitor state Δ^{abs} such that $\Delta^{abs}(T(2, 3))|_+ \neq \Delta^{abs}_0(T(2, 3))|_+$ results in $\ell^+_{\Delta^{abs},q}$ being: $\Delta^{abs}(T(2, 3))|_+ \cup \{\langle T, x_1 = 2 \wedge x_2 = 4 \rangle, \langle R, \top \rangle, \langle S, x_1 = 3 \rangle\}$.

Under-approximating queries. Producing useful lower bounds for queries is more difficult than finding upper bounds. In particular, computing non-trivial lower bounds for a query q is, in general, as difficult as determining whether q is unsatisfiable. Here, we target a restricted class of queries satisfying specific syntactic properties.

We say that a query q is *well-formed* if it is a boolean combination of formulae $\exists\overline{x}.\ T(\overline{x}) \wedge \psi$ such that (1) $nf(T(\overline{x}) \wedge \psi)$ holds, (2) the formula ψ is satisfiable, (3) for any two distinct sub-formulae $\exists\overline{x}.\ T(\overline{x}) \wedge \psi$ and $\exists\overline{x}.\ T(\overline{x}) \wedge \psi'$, there is no $\overline{v}$ satisfying both ψ and ψ', and (4) there are no integrity constraints involving tables occurring in q. The first requirement ensures that we can precisely extract symbolic tuples using the $cstrs(q)$ function described above. The second requirement ensures that each symbolic tuple represents at least one concrete tuple. The third requirement ensures that the symbolic tuples represent disjoint sets of concrete tuples. The fourth requirement, finally, guarantees that integrity constraints do not affect the symbolic tuples in $cstrs(q)$. These requirements guarantee that $cstrs(q)$ correctly identifies a set of tuples that belong to q's support.

For a well-formed query q, we compute the under-approximation $\ell^-_{\Delta^{abs},q}$ as $\bigcup_{\langle T,\varphi\rangle \in cstrs(q)} \left(\bigcup_{T(\overline{v})\in\gamma(\langle T,\varphi\rangle)} \Delta^{abs}(T(\overline{v}))|_-\right)$. If q is not well-formed, then $\ell^-_{\Delta^{abs},q} = \emptyset$. Finally, if q refers to views, then $\ell^-_{\Delta^{abs},q} = \ell^-_{\Delta^{abs},q'}$, where q' is the query obtained by recursively replacing views with their definitions.

Consider the query q: $S(1,2) \vee \neg\exists x.\ T(1,x) \vee \exists x.\ T(2,x)$. This query satisfies our well-formedness criteria. For instance, the two sub-formulae $\exists x.\ T(1,x)$ and $\exists x.\ T(2,x)$ depend on disjoint sets of tuples in the table T. Computing $cstrs(q)$ results in the set $\{\langle S, x_1 = 1 \wedge x_2 = 2\rangle, \langle T, x_1 = 1\rangle, \langle T, x_1 = 2\rangle\}$. Hence, $\ell^-_{\Delta^{abs},q}$ is $\Delta^{abs}(S(1,2))|_- \cup \bigcup_{v\in Val} \Delta^{abs}(T(1,v))|_- \cup \bigcup_{x\in Val} \Delta^{abs}(T(2,v))|_-$.

Labeling users. For the abstract mapping from users to labels, we first define $auth^{abs}$ and afterwards derive $L^{abs}_{\mathcal{U}}$. We use the former to derive an abstract label representing what a user can read according to an arbitrary policy. We use the latter to derive a user's permissions with respect to the current and the initial policies.

Let sec be a security policy and $u \in UID$ be a user. The mapping $auth^{abs}(sec, u)$ assigns to u all the symbolic tuples that can be derived from the tables and views in the policy sec. Observe that we consider only the views in normal form, since those are the only ones that can be directly represented as symbolic tuples, and ignore the others. That is, $auth^{abs}(sec, u)$ is $\langle\{\langle T, \top\rangle \mid T \in auth(s,u) \cap \mathbb{T}\} \cup \{\langle T, \varphi\rangle \mid V \textit{ is a view} \wedge V \in auth(s,u) \wedge def(V) = (T(\overline{x}) \wedge \varphi) \wedge nf(def(V))\}, ST_D\rangle$.

In contrast, the abstract mapping $L^{abs}_{\mathcal{U}}$ from system states and users to labels is as follows. For the attacker atk, $L^{abs}_{\mathcal{U}}(s, atk)$ is the join of what the attacker can read under the current policy sec and the initial policy sec_0, i.e., $auth^{abs}(sec_0, atk) \sqcup^{abs} auth^{abs}(sec, atk)$. For the public user $public$, $L^{abs}_{\mathcal{U}}(s, public) = L^{abs}_{\mathcal{U}}(s, atk)$ since the attacker also observes public observations. Finally, for users u distinct from atk and $public$, $L^{abs}_{\mathcal{U}}(s,u) = \langle ST_D, ST_D\rangle$. Observe that the upper bounds for $auth^{abs}(sec, u)$ and $L^{abs}_{\mathcal{U}}(s, atk)$ are always ST_D, i.e., they represent the $\top$ element in the disclosure lattice. This does not affect our monitor's precision since both $auth^{abs}(sec, u)$ and $L^{abs}_{\mathcal{U}}(s, atk)$ only occur on the left-hand side of $\sqsubseteq^{abs}$, so their upper-bound components are never used.

Consider a policy sec where the user u is authorized to read the table T and the views V (defined as $\{x, y \mid T(x, y) \wedge R(x)\}$) and W (defined as $\{x, y \mid S(x, y) \wedge x \neq y\}$). The function $auth^{abs}$ maps sec and u to the label $\langle\{\langle T, \top\rangle, \langle S, x_1 \neq x_2\rangle\}, ST_D\rangle$. Observe that the view V has been ignored in $auth^{abs}(sec, u)$ since it cannot be under-approximated using symbolic tuples.

Initial monitor state. The initial abstract state Δ^{abs}_0 is as follows: for all predicate queries $T(\overline{v}) \in RC^{pred}$, the initial label $\Delta^{abs}_0(T(\overline{v}))$ corresponds exactly to the query itself, i.e., $\langle\{\langle T, \bigwedge_{1\leq i\leq|T|} x_i = v_i\rangle\}, \{\langle T, \bigwedge_{1\leq i\leq|T|} x_i = v_i\rangle\}\rangle$, whereas for all $x \in Var \cup \{\mathtt{pc}_u \mid u \in UID\}$, the initial label is the one containing no information, i.e., $\Delta^{abs}_0(x) = \langle\emptyset, \emptyset\rangle$.

Soundness. In [23], we prove that the above approximation preserves the monitor's security guarantees. In the next section, we implement this approach in DAISY and evaluate it through different case studies.

VII. Implementation and case studies

We first present DAISY, a security monitor for database-backed SCALA programs. Afterwards, we evaluate our approach's feasibility using four realistic case studies.

A. Securing SCALA programs

We now present DAISY (publicly available at [22]), a security monitor for database-backed SCALA programs, which implements the monitor presented in §V with the approximation from §VI. DAISY enforces end-to-end security across application-database boundaries while supporting advanced database features like triggers and dynamic security policies.

Implementation. We implement DAISY via monitor inlining [16] using SCALA's macro facilities [13]. This allows a programmer to write normal SCALA code that will then be augmented with information-flow checks for both application-level code and database queries simply by adding a `@daisy` annotation on a class, object, or function definition. DAISY uses the Z3 SMT solver [20] to compare symbolic tuples.

Supported fragment. To match the monitor presented in §V, DAISY handles only the imperative subset of SCALA (including all WHILESQL's features) with limited support for higher-order functions. To express queries, DAISY relies on the query language supported by WHILESQL, and it translates queries into SQL commands. The scheduling of threads is currently handled explicitly using the designated function `asUser`. DAISY can easily be extended to directly use SCALA's multi-threading facilities. We refer the reader to DAISY's documentation for a precise definition of the supported fragment.

Extensions. DAISY extends our monitor from §V with configuration functions, and multi-table symbolic tuples.

DAISY allows database administrators to specify functions that modify the database configuration. These functions are annotated with the `@configuration` annotation, and users can invoke them inside their code. These functions also receive as input the identifier of the user invoking them. To avoid leaks, DAISY enforces the following restrictions: (a) functions

annotated with `@configuration` can be executed only when $\Delta(\texttt{pc}) = \bot$, and (b) they can only execute `GRANT`, `REVOKE`, and `CREATE` commands.

DAISY implements a simple generalization of symbolic tuples that allows us to track dependencies across multiple tables, such as those introduced when joining tables. In addition to symbolic tuples of the form $\langle T, \varphi \rangle$, DAISY supports symbolic tuples of the form $\langle \mathbb{T}, \varphi \rangle$, where $\mathbb{T} = T_1 \cdot \ldots \cdot T_n$ is a sequence of table identifiers and φ is a boolean combination of equality and inequality constraints over $T_1 \times \ldots \times T_n$. Informally, $\langle T_1 \cdot \ldots \cdot T_n, \varphi \rangle$ represents a set of concrete tuples over the Cartesian product of the tables $T_1, \ldots, T_n$. Here, we discuss how we extend $\sqsubseteq^{abs}$ to handle multi-table symbolic tuples. The other operators are extended in a straightforward way. Given two labels $\langle S_1^-, S_1^+ \rangle$ and $\langle S_2^-, S_2^+ \rangle$, $\langle S_1^-, S_1^+ \rangle \sqsubseteq^{abs} \langle S_2^-, S_2^+ \rangle$ iff for all symbolic tuples $\langle \mathbb{T}, \varphi \rangle \in S_1^+$, there are symbolic tuples $\langle \mathbb{T}_1, \varphi_1 \rangle, \ldots, \langle \mathbb{T}_n, \varphi_n \rangle$ in S_2^- such that $\mathbb{T} = \mathbb{T}_1 \cdot \ldots \cdot \mathbb{T}_n$ and $\varphi \models \varphi'_1 \wedge \ldots \wedge \varphi'_n$ (where φ'_i is obtained from φ_i by renaming x_j as $x_{j+\sum_{i<j}(\mathbb{T}_i|_0 + \ldots + \mathbb{T}_i|_{|\mathbb{T}_i|})}$).

B. Case studies

To evaluate DAISY, we carried out four case studies (available at [22]): (i) a social network, (ii) an assignment grading system, (iii) a calendar application, and (iv) a conference management system. Note that we only focus on the security-critical parts of the applications. Our evaluation has three objectives: to (1) validate that DAISY provides the desired security guarantees, (2) confirm that our approximation is not overly restrictive, and (3) evaluate DAISY's overhead.

1) Social network: We implemented in SCALA the social network model from §II. Without the trigger, DAISY considers the program from §II as secure, since there is no leak of sensitive information. When the trigger is in place, DAISY correctly identifies the leak of sensitive information. Specifically, by leveraging our expansion procedure, DAISY successfully tracks the flows of information across the program-database boundaries and correctly rejects the program as insecure. Existing approaches ignore the leaks caused by triggers and would accept the program as secure. Moreover, our approximation is sufficiently precise to correctly enforce the row-level policy "each user can read only his friends' reviews"; which cannot be enforced by existing approaches that track column-level dependencies.

2) Assignment grading system: We model a system inspired by one of URFLOW's case studies [14]. The system allows students to hand-in assignments that are graded by teaching assistants (TAs) who only have access to students' pseudonyms.

Database schema. The table `students` holds the students' data. The table `codes` maps students to their pseudonyms. The table `tas` stores TAs' names, and `handins(ID, txt)` records student submissions. The table `grades(ID, grade)` stores the hand-ins' grades, and `owner(ID, studID)` associates the hand-ins with pseudonyms.

Security policy. Students are authorized to read their own pseudonym, but they cannot read other entries in the table `codes`. Moreover, they can read the grades only of their own submissions. In contrast, TAs can read the `handins` table and can read and modify the `grades` table. Thus, according to our policy, a TA cannot leak information about a student s to another student. We implement this policy using views and `GRANT` commands; see [22].

Examples. In the following, a student submits a hand-in, a TA grades it, and, then, the same student reads the grade.

```
asUser("stud1") {submitHandin("stud1",
    "GoodSubmission")}
// TA inspects submission and grades it
asUser("ta") {
  val firstSubmission = viewSubmissions().head
  outputTo("ta", firstSubmission)
  grade(firstSubmission, "Good")
} // student reads the grade:
asUser("stud1") { viewGrade("stud1") }
```

The example uses the helper functions `submitHandin`, `grade`, `viewGrade`, and `viewSubmissions`, which encapsulate the interaction with the database. For example, the `viewSubmissions` function is as follows:

```
def viewSubmissions() = select("{id, text |
    handins(id, text)}")
```

DAISY accepts this program as secure and successfully enforces the row-level policy "each student can read his grades." UR/FLOW would also consider the above program as secure.

Now, consider the same program where the function `viewSubmissions` is defined as `select("{id, text | handins(id, text)AND codes('stud1', 'xyz'}")`. The program violates our policy: observing the output of `viewSubmissions` leaks information about `codes` to the TA. DAISY correctly detects such a leak and rejects the program as insecure. UR/FLOW, however, would accept the program as secure, since it ignores implicit leaks introduced by queries [14].

Finally, the TA tries to output the grades to a student *stud2*. DAISY prevents this since `grades` contains information about *stud1* that should not flow to *stud2*.

```
asUser("ta") { // TA tries to leak everything:
 val gr = select("{id, gr | grades(id, gr)}")
 outputTo("stud2", gr) }
```

3) Calendar: We implement a calendar application that supports creating events and adding other users as attendees. We use DAISY to enforce the following policy: each user u can read the information about an event's participants only if u is attending the event. As a result, if the event's organizer removes an attendee, that attendee can no longer view the event's other attendees. We implement the calendar application as well as examples that comply with and violate the above policy. See [22] for further details.

4) Conference management system: We model the key aspects of a conference management system.

Database schema. The table `user(ID, name)` holds the users' data. The table `paper(paperID, confID, title)` stores the papers' information, whereas the table `authors(paperID, authorID)` maps papers to authors and `reviewer(confID, revID)` associates conferences with

reviewers. The table $\texttt{review}(\texttt{paperID}, \texttt{revID}, \texttt{decision})$ stores reviews' information.

Security policy. In our system, we have two roles: reviewers and authors. As an author, a user u can access only the reviews of his own papers. To encode this, for each user u, we introduce the view $\texttt{review}^{\texttt{A}}_u = \{p, r, d \mid \texttt{review}(p, r, d) \wedge \texttt{author}(p, u)\}\}$. As a reviewer, a user u can access the reviews of all papers submitted to conferences where he is a reviewer. This is implemented using the view $\texttt{review}^{\texttt{R}}_u = \{p, r, d \mid \texttt{review}(p, r, d) \wedge \exists c, t.\ (\texttt{paper}(p, c, t) \wedge \texttt{reviewer}(c, u))\}$. We can now define the permissions. Whenever a user u acts as author, he can read $\texttt{review}^{\texttt{A}}_u$. In contrast, when a user u acts as reviewer, he can read $\texttt{review}^{\texttt{R}}_u$. Moreover, users can always read the tables `user`, `author`, and `reviewer`. We model users logging in as authors or as reviewers using the configuration functions `asAuthor` and `asReviewer`, which are executed under the administrator's privileges and modify the policy as expected.

Examples. In the following snippet, a user u logs into the application as an author (modeled using the `asAuthor` function) and retrieves the reviews of his EuroS&P papers.

```
asAuthor()
val revs = extractReviews("u", "EuroS&P 2019")
outputTo("u", revs)
```

This example relies on the `extractReviews` helper function, which returns the result of the query `SELECT` $\{p, t, d \mid \texttt{reviews}(p, c, t, d) \wedge \texttt{author}(p, c, u)\}$, where u and c are the user and the conference given as input. Symbolic tuples are precise enough to determine that *revs*' content depends only on authorized information. Hence, DAISY correctly accepts this program as secure. Approaches based on column-level dependencies would reject this program as insecure.

To illustrate dynamic policies, consider the following snippet, where a user u logs in as a reviewer, stores all reviews of all papers for the conferences where he is a program committee member in a variable *data*, switches his role to author, and prints the data.

```
asReviewer()
val data = conferenceData("u")
asAuthor()
outputTo("u", data)
```

This example uses the `conferenceData` helper function that returns the result of the query `SELECT` $\{p, t, d \mid \texttt{review}(p, c, t, d) \wedge \texttt{reviewer}(c, u)\}$, where u is the user given as input. The example violates our policy. While the function `conferenceData` accesses only authorized data when u is logged as a reviewer, the information is disclosed only after the privileges have been revoked. DAISY detects that *data*'s content is no longer authorized in the last statement and correctly stops the execution. Hence, DAISY correctly handles dynamic policies and tracks dependencies across policy changes.

5) Performance: We benchmarked our case studies (each one comprising roughly 100 lines of code) on a 64-bit i7-4600U CPU running ArchLinux with OpenJDK version 1.8.0_144. In our experiments, DAISY introduces an overhead between 5% and 10% compared to the code's unmonitored execution. We believe is acceptable for a proof-of-concept implementation.

VIII. Related work

IFC for database-backed applications. We compare our work with existing IFC solutions for database-backed applications [7], [14], [15], [17], [28], [31], [44], [49] with respect to three aspects: (1) the database model, (2) the supported security policies, and (3) whether the solution has been proved sound. Figure 5 summarises how existing approaches fare with respect to these criteria.

SIF [15] enforces IFC policies for Java web applications, whereas Li and Zdancewic [31] present a system for statically checking IFC policies for database-backed applications. Both approaches are type-based, require programmers to manually annotate programs with typing annotations, and consider only simple database models and column-level policies. Another type-based approach is IFDB [44], a system supporting decentralized IFC across databases and applications. Its *Query by Label* model extends work on multi-level secure (MLS) databases [33] and provides abstractions for dealing with expressive IFC policies. It supports complex database features and policies. Similarly to other MLS approaches, it relies on poly-instantiation [30], which is not supported by the SQL standard and requires ad-hoc extensions [21], [42]. Moreover, it has neither a formal semantics nor a soundness proof. In contrast to these type-based approaches, we do not require program annotations, we support more complex dynamic row-level policies, and our solution comes with a soundness proof of security for a realistic database model.

JSLINQ [7], SELINKS [17], [45], and SELINQ [43] secure applications that interact with databases through language-integrated queries. In contrast to DAISY, they consider simpler database models and ignore constructs like triggers and integrity constraints. Moreover, JSLINQ and SELINQ only support column-level policies, while SELINKS also supports row-level policies. However, none of them support row-level policies where privileges can be granted and revoked as we do. Lourenço and Caires [32] introduce dependent information flow types which allow the types' security levels to depend on runtime values, thus enabling row-level policies. Their main goal is using dependent types for IFC; they therefore ignore the challenges posed by advanced database features and dynamic policies.

URFLOW [14] is a static information flow analysis tool for UR/WEB applications. It supports policies expressed as SQL queries that leverage the users' runtime knowledge. The enforcement is done by symbolic execution over a model of the web application. DAISY can enforce similar policies and it supports features like triggers and dynamic policies. Moreover, URFLOW provides no precise security guarantees, as it ignores some implicit flows.

LWEB [36] is a framework for developing secure multi-tier applications in Haskell. LWEB enforces data-dependent column- and row-level policies (expressed in Haskell), where the labels associated with columns and tuples may depend on the tuples' values. Similarly to LWEB, we also support

	Database Features					Security policies		Soundness proof
	`INSERT - DELETE`	Dynamic policies	Triggers	Integrity constraints	Views	Column level	Row level	
SIF [15]	✓	✓[1]				✓		✓
Li et al. [31]		✓[1]				✓		
IFDB [44]	✓	✓[1]	✓	✓	✓	✓	✓	
JsLinq [7]		✓[1]				✓		✓
SELinks [17]	✓	✓[1]				✓	✓	✓
SeLINQ [43]						✓		✓
Lourenço et al. [32]	✓					✓	✓	✓
UrFlow [14]	✓					✓	✓	
LWeb [36]	✓	✓[1]				✓	✓	✓
Jacqueline [49]	✓	✓				✓	✓	✓
Daisy	✓	✓	✓	✓	✓	✓[2]	✓	✓

[1]Only declassification [2]Only static column-level policies

Fig. 5: Comparison with other IFC approaches for database-backed applications

data-dependent row-level policies, which can be formalized using views, and a restricted class of column-level policies. In contrast to our work, LWeb ignore advanced database features, like triggers, and it supports only declassification, while Daisy supports dynamic policies where permissions can be granted and revoked at runtime.

Jacqueline [49] presents an IFC approach that secures database-backed applications using faceted execution [5]. Jacqueline adopts a policy-agnostic programming model, where the language runtime modifies the computation to produce policy compliant results. In contrast to modifying the results, our monitor prevents leaks by terminating the execution. In Jacqueline, security policies are formalized as program functions and both row-level and column-level policies are supported. However, Jacqueline consider a simpler database model than our work and it ignores security-critical database features like triggers.

To summarise, existing works consider unrealistic database models, ignore dynamic policies where permissions can be granted and revoked, or provide informal soundness arguments. In contrast, our work has the following distinguishing features: (1) a realistic database model, which accounts for security-critical constructs like triggers, views, and dynamic policies, (2) a monitor combining information-flow tracking with disclosure lattices that can enforce dynamic row-level and static column-level policies, and (3) a soundness proof of security for a realistic database model.

Security conditions. Our security condition is inspired by existing knowledge-based notions for dynamic policies [3], [6], [11]. While the semantics for dynamic policies remains an open research problem, our security condition captures security with respect to a perfect recall attacker. Askarov and Chong [3] propose security conditions against all attackers. We conjecture that our security monitor also enforces security against all attackers. Hicks et al. [27] propose *non-interference between updates*, which ensures non-interference between policy changes, while ignoring information leaks across such changes. Bohannon et al. [10] study *reactive noninterference* to reason about security policies in languages with event handlers like client-side web applications. The execution model for event handlers is similar to the execution of triggers in our language. We refer the reader to Broberg et al. [11] for a survey of dynamic policies.

Label models. The *universal lattice* by Hunt and Sands [29] allows expressing dependencies between variables, where the lattice's elements are sets of variables and the order relationship is set containment. In contrast, disclosure lattices allow us to reason about dependencies between queries. By directly combining disclosure lattices with dynamic information-flow tracking, we track tuple-level dependencies between variables and queries, which would otherwise be lost using simpler label models, e.g., the "high" and "low" lattice. This allows us to support dynamic row-level policies and static column-level policies.

Database access control. Many security conditions have been proposed for attackers that can issue only `SELECT` queries [8], [9], [24], [37], [48]. Guarnieri et al. [25] extend database access control by supporting advanced features, such as triggers and dynamic policies. WhileSQL's database model builds on top of Guarnieri et al.'s database semantics. Bender et al. [8], [9] introduce disclosure lattices to reason about fine-grained security policies in databases. We leverage disclosure lattices to track information-flows through the application and database boundary.

Qapla [34] is a database access control middleware supporting complex security policies, such as linking and aggregation policies, that go beyond what is supported by commercial database systems. Our monitor supports only policies that can be expressed in the SQL access control model. Hence, it does not support policies like linking or aggregation. Qapla, however, cannot enforce end-to-end IFC policies across the application/database boundary.

Research on mandatory database access control has historically focused on Multi-Level Security [21], [33], where both the data and the users are associated with security levels. In contrast to WhileSQL, MLS systems consider, in general, fixed security policies (cf. the *tranquility principle* [41]) and rely on poly-instantiation [30].

IX. Conclusion

Securing database-backed applications requires reasoning about the program and the database as a whole. Motivated by

the severe limitations of existing approaches, we developed a novel security monitor that enforces security policies in an end-to-end fashion across the application-database boundary. In contrast to existing approaches, our monitor accounts for realistic database model, and it leverages disclosure lattices to track fine-grained tuple-level dependencies between variables and tuples and to enforce expressive dynamic policies. DAISY implements our security monitor for SCALA programs, and it relies on symbolic tuples, a novel efficient approximation of disclosure lattices. DAISY demonstrates how realistic database models and database theory can be combined with language-based security techniques to effectively protect systems against larger classes of attacks.

ACKNOWLEDGEMENTS

This work was partly funded by the Swedish Foundation for Strategic Research (SSF) under the projects WebSec and TrustFull, by the Swedish Research Council (VR) under the projects PrinSec and JointForce, by the Madrid regional project 2018-T2/TIC-11732, and by a grant from the Intel Corporation.

REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of databases*. Addison-Wesley Reading, 1995, vol. 8.
[2] O. Arden, J. Liu, and A. C. Myers, "Flow-limited authorization," in *CSF*, 2015.
[3] A. Askarov and S. Chong, "Learning is change in knowledge: Knowledge-based security for dynamic policies," in *CSF*, 2012.
[4] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, "Termination-insensitive noninterference leaks more than just a bit," in *ESORICS*, 2008.
[5] T. H. Austin and C. Flanagan, "Multiple facets for dynamic information flow," in *POPL*, 2012.
[6] M. Balliu, "A logic for information flow analysis of distributed programs," in *NordSec*, 2013.
[7] M. Balliu, B. Liebe, D. Schoepe, and A. Sabelfeld, "Jslinq: Building secure applications across tiers," in *CODASPY*, 2016.
[8] G. Bender, L. Kot, and J. Gehrke, "Explainable security for relational databases," in *SIGMOD*, 2014.
[9] G. Bender, L. Kot, J. Gehrke, and C. Koch, "Fine-grained disclosure control for app ecosystems," in *SIGMOD*, 2013.
[10] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic, "Reactive noninterference," in *CCS*, 2009.
[11] N. Broberg, B. van Delft, and D. Sands, "The anatomy and facets of dynamic policies," in *CSF*, 2015.
[12] K. Browder and M. Davidson, "The virtual private database in Oracle9iR2," *Oracle Technical White Paper, Oracle Corporation*, vol. 500, 2002.
[13] E. Burmako, "Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming," in *SCALA@ECOOP*, 2013.
[14] A. Chlipala, "Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications." in *OSDI*, 2010.
[15] S. Chong, K. Vikram, and A. C. Myers, "SIF: Enforcing Confidentiality and Integrity in Web Applications," in *USENIX Security*, 2007.
[16] A. Chudnov and D. A. Naumann, "Information flow monitor inlining," in *CSF*, 2010.
[17] B. J. Corcoran, N. Swamy, and M. W. Hicks, "Cross-tier, label-based security enforcement for web applications." in *SIGMOD*, 2009.
[18] B. A. Davey and H. A. Priestley, *Introduction to lattices and order*. Cambridge university press, 2002.
[19] B. Davis and H. Chen, "DBTaint: cross-application information flow tracking via databases," in *WebApps*, 2010.
[20] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*, 2008.
[21] D. E. Denning and T. F. Lunt, "A multilevel relational data model," in *S&P*, 1987.
[22] M. Guarnieri, M. Balliu, D. Schoepe, D. Basin, and A. Sabelfeld, "DAISY: DAtabase and Information-flow SecuritY," https://www.cse.chalmers.se/research/group/security/DAISY, 2019.
[23] ——, "Information-Flow Control for Database-backed Applications – Technical Report," https://www.cse.chalmers.se/research/group/security/DAISY, 2019.
[24] M. Guarnieri and D. Basin, "Optimal security-aware query processing," in *VLDB*, 2014.
[25] M. Guarnieri, S. Marinovic, and D. Basin, "Strong and provably secure database access control," in *EuroS&P*, 2016.
[26] D. Hedin and A. Sabelfeld, "A perspective on information-flow control," in *Software Safety and Security-Tools for Analysis and Verification*, 2012.
[27] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic, "Dynamic updating of information-flow policies," in *FCS*, 2005.
[28] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection." in *WWW*, 2004.
[29] S. Hunt and D. Sands, "On flow-sensitive security types," in *POPL*, 2006.
[30] S. Jajodia and R. Sandhu, "Polyinstantiation integrity in multilevel relations," in *S&P*, 1990.
[31] P. Li and S. Zdancewic, "Practical information flow control in web-based information systems," in *CSF*, 2005.
[32] L. Lourenço and L. Caires, "Dependent information flow types," in *POPL*, 2015.
[33] T. F. Lunt, D. E. Denning, R. R. Schell, M. Heckman, and W. R. Shockley, "The seaview security model," *TSE*, vol. 16, no. 6, 1990.
[34] A. Mehta, E. Elnikety, K. Harvey, D. Garg, and P. Druschel, "Qapla: Policy compliance for database-backed systems," in *USENIX Security*, 2017.
[35] A. Nash, L. Segoufin, and V. Vianu, "Views and queries: Determinacy and rewriting," *TODS*, vol. 35, no. 3, p. 21, 2010.
[36] J. Parker, N. Vazou, and M. Hicks, "LWeb: Information flow security for multi-tier web applications," in *POPL*, 2019.
[37] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy, "Extending query rewriting techniques for fine-grained access control," in *SIGMOD*, 2004.
[38] A. Russo and A. Sabelfeld, "Dynamic vs. static flow-sensitive security analysis," in *CSF*, 2010.
[39] A. Sabelfeld and D. Sands, "Probabilistic noninterference for multi-threaded programs," in *CSFW*, 2000.
[40] P. Samarati, "Recursive revoke," in *Encyclopedia of Cryptography and Security*. Springer, 2011, pp. 1035–1037.
[41] P. Samarati and S. Capitani de Vimercati, "Access Control: Policies, Models, and Mechanisms," *Springer LNCS*, vol. 2171, 2001.
[42] R. Sandhu and F. Chen, "The multilevel relational (MLR) data model," *TISSEC*, vol. 1, no. 1, 1998.
[43] D. Schoepe, D. Hedin, and A. Sabelfeld, "Selinq: Tracking information across application-database boundaries," in *ICFP*, 2014.
[44] D. Schultz and B. Liskov, "IFDB: decentralized information flow control for databases," in *EuroSys*, 2013.
[45] N. Swamy, B. J. Corcoran, and M. Hicks, "Fable: A language for enforcing user-defined security policies," in *S&P'08*, 2008.
[46] B. van Delft, S. Hunt, and D. Sands, "Very static enforcement of dynamic policies," in *POST*, 2015.
[47] S. D. C. d. Vimercati and G. Livraga, "SQL access control model," in *Encyclopedia of Cryptography and Security*. Springer, 2011.
[48] Q. Wang, T. Yu, N. Li, J. Lobo, E. Bertino, K. Irwin, and J.-W. Byun, "On the correctness criteria of fine-grained access control in relational databases," in *VLDB*, 2007.
[49] J. Yang, T. Hance, T. H. Austin, A. Solar-Lezama, C. Flanagan, and S. Chong, "Precise, dynamic information flow for database-backed applications," in *PLDI*, 2016.
[50] S. A. Zdancewic, "Programming languages for information security," Ph.D. dissertation, Cornell University, Ithaca, NY, USA, 2002.