Information-Flow Control for Database-backed Applications

Abstract—**Securing database-backed applications requires tracking information across the 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. Previous work, 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. The 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 the monitor for SCALA programs and demonstrate its effectiveness on four case studies.

1. 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 security of the overall system. 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 [20], [18], [9], [30], [48], [43], [15], [16] extend programs with database models and apply standard IFC techniques, such as security type systems [42], [18], symbolic execution [15], or faceted values [48], 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 database systems and can be exploited to violate the database's confidentiality [24]. Ignoring them, therefore, means ignoring possible information leaks.

Another challenge, when it comes to tracking information flows across the program-database boundary, is analyzing queries. Some approaches [42], [9] 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 database theory concepts.

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. [24] 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. The monitor enforces security by tracking fine-grained dependencies between variables and queries across program-level computations and blocking outputs that could potentially leak sensitive information. For checking policy violations, the monitor relies on disclosure lattices [10] and query determinacy [34]. The monitor supports row-level policies, a common class of database policies used in many fine-grained access control models [13], [47], [36], [23]. Additionally, it supports security-critical database features, such as triggers and policy changes, that are not supported by existing mechanisms [20], [42], [18], [30], [48], [43]. 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 the undecidability issues over disclosure lattices, DAISY precisely approximates them using symbolic tuples. We demonstrate the precision and feasibility of our approach 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 between 5% and 10% in our case studies.

2. 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 tool, 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 the access permissions of each user account for the database tables and views. The system does not include an access control mechanism that enforces *read* permissions over tables and views. However, we assume that the system correctly enforces *write* permissions, e.g., a user can insert a tuple in a table T only if the policy says so. This allows us to study what it means for a system to be secure, independently of the enforcement. That is, 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.

Setting. We consider a *social network* allowing users to review books, publish their reviews, and share them with friends. The database contains 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 $\texttt{SELECT} * \texttt{FROM review WHERE userId} = u$.

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 can be implemented 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 [13], [47], [36], [23]. 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 our goal is reasoning about the above policy as an information-flow policy, not as an access control policy.

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.

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 $Carl$
$x \leftarrow$ INSERT INTO review(id, user, book, text, score)
    VALUES (1, $Carl$, "War and Peace", "..", 10)
//Executed by $Alice$
$y \leftarrow$ SELECT revId, text, score FROM review WHERE
    book = "War and Peace" AND userID = $Carl$
**out**($Alice, y$)
$z \leftarrow$ INSERT INTO likes VALUES ($y.revId$,
    "War and Peace", $Carl, Alice$)
//Executed by $Bob$
$f \leftarrow$ SELECT $u_2$ FROM friends WHERE $u_1 = Bob$
$S \leftarrow$ SELECT genre FROM stats WHERE userId = $Bob$
**for** ($fr : f; g : S$)
  $v \leftarrow$ SELECT v FROM stats WHERE userId = $fr$
          AND genre = $g$
  **out**($Bob, \langle fr, g, v\rangle$)
```

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 essentials? The above example relies only on 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, a piece of 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 sum of all reviews' scores for those reviews of books whose genre is g that have been 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, increments the score's counter whenever a new tuple is inserted in the table `likes`.

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

Specifically, whenever someone inserts a tuple $\langle revId, book, revAuthor, user\rangle$ in `likes`, the trigger increments the counter associated with the user $user$ and $book$'s genre by 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.

Our program is no longer secure when the trigger tr is 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 in the table likes, the trigger tr is executed and one of the counters in stats is incremented by 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.

DAISY to the rescue. 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 to account for realistic database models and for common policy idioms used in database security. Unfortunately, existing solutions [20], [18], [9], [30], [48], [43], [15], [16], [42] either ignore relevant security-critical database features (like triggers and dynamic policies) or adopt imprecise analyses when handling queries (cf. §8). 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 does DAISY work? DAISY tracks 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 result and it tracks how the retrieved information flows through the program. To concisely represents sets of tuples, we develop *symbolic tuples*, an efficient approximation of disclosure lattices (cf. §6), 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 $\langle \texttt{review}, \texttt{userId} = Carl \wedge \texttt{book} = \texttt{"War and Peace"}\rangle$, which denotes that y's content depends on the values of all tuples in the table review satisfying the constraint $\texttt{userId} = Carl \wedge \texttt{book} = \texttt{"War and Peace"}$. When $Alice$ inserts a tuple in the table likes, DAISY tracks the information flow caused by the trigger and it determines that some of the values in the table stats now depend on $Carl$'s review. This is done by propagating the label $\langle \texttt{review}, \texttt{userId} = Carl \wedge \texttt{book} = \texttt{"War and Peace"}\rangle$ to the tuples in table stats that store $Alice$'s counters. DAISY also tracks the label $\langle \texttt{review}, \texttt{userId} = Carl \wedge \texttt{book} = \texttt{"War and Peace"}\rangle$ across the computation performed by Bob. Finally, before executing the output statement $\textbf{out}(Bob, \langle Alice, \texttt{novel}, v\rangle)$, DAISY compares $\langle Alice, \texttt{novel}, v\rangle$'s label, which contains, among other labels, the symbolic tuple $\langle \texttt{review}, \texttt{userId} = Carl \wedge \texttt{book} = \texttt{"War and Peace"}\rangle$, with the permissions granted by the current security policy to the user Bob. Concretely, Bob's permissions are also represented using symbolic tuples. In particular, the labels $\langle \texttt{book}, \top\rangle$, $\langle \texttt{user}, \top\rangle$, $\langle \texttt{friends}, \top\rangle$, $\langle \texttt{likes}, \top\rangle$, and $\langle \texttt{stats}, \top\rangle$ denote that Bob has full access to the tables book, user, friends, likes, and stats (the constraint $\top$ is trivially satisfied by all concrete tuples). Moreover, there is one label $\langle \texttt{review}, \texttt{authorId} = u\rangle$ for each user u who is Bob's friend.

When comparing labels, DAISY checks whether $\langle Alice, \texttt{novel}, v\rangle$'s content depends on data that is authorized with respect to the security policy. Using query determinacy, DAISY checks if the symbolic tuples associated with $\langle Alice, \texttt{novel}, v\rangle$ can be derived from those associated with Bob's permissions. Since Bob cannot access $review_{Carl}$, there is no symbolic tuple among Bob's permissions that discloses the information represented by $\langle Alice, \texttt{novel}, v\rangle$' label $\langle \texttt{review}, \texttt{userId} = Carl \wedge \texttt{book} = \texttt{"War and Peace"}\rangle$. Hence, DAISY stops the program and prevents the leak of sensitive information.

Organization. We formalize WHILESQL in §3 and our security condition in §4. We present our monitor in §5 and symbolic tuples in §6. We present DAISY and our case studies in §7, we discuss related work in §8, and we draw conclusions in §9. Figure 6 summarizes the notation used in the paper. A technical report with complete proofs of all results is available at [3], while DAISY is available at [2].

3. WHILESQL

3.1. Syntax and notation

Syntax. WHILESQL is an imperative language with querying capabilities, whose syntax is in Figure 7. Its imperative fragment consists of assignments, conditionals, loops, and output statements $\textbf{out}(u, e)$, which print the value of an expression e to a user u. 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 not allowed by the current security policy or 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. [24], which supports security-critical features like dynamic policies and triggers. Hence, following [24], we make various simplifications to our query language.

WHILESQL supports the retrieval of 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].

WHILESQL allows changes to the database's content using INSERT and DELETE commands. Specifically, we support INSERT and DELETE commands that explicitly

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

Figure 1: WHILESQL's syntax

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. Note that 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 [46], [39].

Our model also supports triggers, which are procedures automatically executed by the database system in response to user commands. In particular, we support only `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 severe 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 a `GRANT`, `REVOKE`, `INSERT`, or `DELETE` command. Following [24], 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.

Sequences. 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.

Users. The set $\mathcal{U}$ of all users is $UID \cup \{public\}$, where UID is a set of user identifiers and $public$ is a designated user identifier.

3.2. Local semantics

We define here the semantics of WHILESQL programs executed in isolation. 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 configuration $M = \langle D, \Gamma \rangle$.

Following [24], we define a *security policy* to be a finite set of `GRANT` statements. Given a policy sec and a user u, we denote by $auth(sec, u)$ 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. Note that 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 [24] 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.

A *memory* $m \in Mem$ is a function mapping variables to values, i.e., $Mem = Var \to Val$. 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 $\langle s, ctx \rangle \in \Omega_M$. A configuration is *initial* iff $ctx = \epsilon$.

In WHILESQL, there are two ways of producing observations. First, $\textbf{out}(u, e)$ statements can be used to output information to users. Second, successfully executing `GRANT`, `REVOKE`, and `CREATE` commands produces 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 a user and o is a value in Val or a `GRANT`, `REVOKE`, or `CREATE` command, and Obs denotes the set of all observations.

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

E-QUERYOK

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

E-QUERYEX

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

Figure 2: Rules handling the query's execution

respects the rules defining $\rightarrow_u$. 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 $\mathit{trace}(r)$, is obtained by concatenating all observations in the run.

We rely on [24] for the semantics of SQL statements. Our operational semantics uses the function $[\![q]\!](\langle s, \mathit{ctx}\rangle, u)$ (defined in Appendix E) to connect the WHILESQL's semantics with the database's semantics. The function $[\![q]\!](\langle s, \mathit{ctx}\rangle, u)$ takes as input an SQL command q, a runtime state $\langle s, \mathit{ctx}\rangle \in \Omega_M$, and the user $u \in \mathit{UID}$ executing the command, and it returns a tuple $\langle\langle s', \mathit{ctx}'\rangle, r, \mathit{em}\rangle$, where $\langle s', \mathit{ctx}'\rangle \in \Omega_M$ 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 regulating the query 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 $[\![q]\!](\langle s, \mathit{ctx}\rangle, u)$) and it stores the query's result in the memory. The rule relies on the function $\mathit{obs}(q)$, which takes as input a query q, to conditionally produce a public observation $\langle \mathit{public}, q\rangle$ in case the command q modifies the database configuration. Formally, $\mathit{obs}(q) = \langle \mathit{public}, q\rangle$ in case 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 full details are given in Appendix E.

3.3. Global semantics

We denote the set of commands together with the executing user by $\mathit{Com}_{\mathit{UID}} = \mathit{UID} \times \mathit{Com}$ and the set of pairs of users and memories as $\mathit{Mem}_{\mathit{UID}} = \mathit{UID} \times \mathit{Mem}$. 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 $\langle C, M, \langle s, \mathit{ctx}\rangle, \mathcal{S}\rangle \in \mathit{GlConf}$, where $C \in \mathit{Com}^*_{\mathit{UID}}$ is a sequence of WHILESQL programs with the executing users, $M \in \mathit{Mem}^*_{\mathit{UID}}$ is a sequence of memories, $\langle s, \mathit{ctx}\rangle \in \Omega_M$ 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 $\langle C, M, \langle s, \mathit{ctx}\rangle, \mathcal{S}\rangle$ 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 $M \in \mathit{Mem}^*_{\mathit{UID}}$ and s is a system state. Our global semantics is standard and we formalize it in Appendix E. 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 either to tables in the database schema or to previously created views.

4. 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 [3], we consider the more general case where the memory content may be sensitive.

4.1. Preliminaries

Memory equivalence. Two sequences of memories M_1, $M_2 \in \mathit{Mem}^*_{\mathit{UID}}$ are equivalent for a user u iff they have the same length, they involve the same users, and the memory associated with u is identical in M_1 and M_2. Formally, M_1 and M_2 are *u-equivalent*, written $M_1 \approx_u M_2$, iff $|M_1| = |M_2|$ and for all $1 \leq i \leq |M_1|$, $u_1^i = u_2^i$ and if $u_1^i = u$, then $m_1^i = m_2^i$, where $M_j|_i = \langle u_j^i, m_j^i\rangle$ for $j \in \{1, 2\}$.

Database equivalence. Given two database states db and db' and a set S of tables and views, $\mathit{db} \approx_S \mathit{db}'$ 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 [24]. 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 $s = \langle \mathit{db}, U, \mathit{sec}, T, V\rangle$ and $s' = \langle \mathit{db}', U', \mathit{sec}', T', V'\rangle$ are *u-equivalent*, written $s \approx_u s'$, iff (1) $U = U'$, (2) $\mathit{sec} = \mathit{sec}'$, (3) $T = T'$, (4) $V = V'$, and (5) $\mathit{db} \approx_{\mathit{auth}(\mathit{sec}, u)} \mathit{db}'$.

Global state equivalence. Two global states $\langle M, s\rangle$ and $\langle M', s'\rangle$ are *u-equivalent* for a user u, written $\langle M, s\rangle \approx_u \langle M', s'\rangle$, iff $M \approx_u M'$ and $s \approx_u s'$. We denote by $[\langle M, s\rangle]_{\approx_u}$ the set $\{\langle M', s'\rangle \mid \langle M', s'\rangle \approx_u \langle M, s\rangle\}$ of global states that are u-equivalent to $\langle M, s\rangle$.

Trace equivalence. To formalize equivalence between traces, we first define the *projection of a trace τ for a user u*, written $\tau{\upharpoonright_u}$, as follows: $\varepsilon{\upharpoonright_u} = \varepsilon$, $(\langle u', o\rangle \cdot \tau'){\upharpoonright_u} = \langle u', o\rangle \cdot \tau'{\upharpoonright_u}$ if $u' = \mathit{public}$ or $u' = u$, and $(\langle u', o\rangle \cdot \tau'){\upharpoonright_u} = \tau'{\upharpoonright_u}$ otherwise. 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., $\tau_1{\upharpoonright_u} \preceq \tau_2{\upharpoonright_u}$ or $\tau_2{\upharpoonright_u} \preceq \tau_1{\upharpoonright_u}$.

4.2. Knowledge

Following [5], [45], we characterize what a user can infer from an execution in terms of his *knowledge*, i.e., the set of global 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 $\{\langle M, s\rangle \mid \langle M, s\rangle \approx_u \langle M_0, s_0\rangle \wedge \forall ctx', \tau', C', M', s', \mathcal{S}'. (\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau'}{}^{*} \langle C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle \Rightarrow \tau \sim_u \tau')\}$.

A user u's knowledge is the set of initial global states that u considers possible after having observed $\tau\upharpoonright_u$. 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/divergence an observable event [6], [25], while other definitions, in line with ours, do not [5], [45], thus ignoring *pure* progress leaks, i.e., progress leaks not related to divergence/termination.

4.3. 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 [5], [12].

We interpret security policies with respect to initial global states. The *allowed knowledge* $A_{u,sec}$ determines the set of initial global states that a user u considers possible for a given policy sec. Given a sequence of memories $M_0 \in Mem^*_{UID}$, 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 $A_{u,sec}(M_0, s_0)$ as $\{\langle M, s\rangle \mid s \approx_{sec,u} s_0 \wedge M \approx_u M_0\}$, 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 $A_{u,sec}(M_0, s_0)$ *allowed* knowledge since it represents the knowledge of the initial global state that the user u is permitted to learn given the policy sec. In contrast to $[\langle M_0, s_0\rangle]_{\approx_u}$, $A_{u,sec}(M_0, s_0)$ contains the global states that agree with $\langle M_0, s_0\rangle$ 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 $\mathcal{S}$, a system state s_0, and a sequence of memories $M_0 \in Mem^*_{UID}$ 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 $1 \leq i \leq n$, $K_u(\langle M_0, s_0\rangle, C, \mathcal{S}, trace(r^{i-1})) \cap A_{u,sec}(M_0, s_0) \subseteq K_u(\langle M_0, s_0\rangle, C, \mathcal{S}, trace(r^i))$, where the database 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.

5. 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), and 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. Moreover, the monitor 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.

5.1. Preliminaries

We leverage *disclosure lattices* to reason about the information disclosed by sets of queries [10]. Recall that a security policy defines 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 §5.3. In the following, we fix a database configuration $\langle D, \Gamma\rangle$.

Predicate queries. A *predicate query* is a query of the form $T(\overline{v})$ where T is a table identifier in D and $\overline{v} \in Val^{|T|}$. A predicate query represents a single tuple in the database. The set of all predicate queries is RC^{pred}.

Determinacy. Query determinacy [34] 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 $D, \Gamma \vdash Q \twoheadrightarrow Q'$, 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 $minDet_{D,\Gamma}(Q, q)$, iff Q is the

$$\begin{tikzcd}
& cl(\{T(1), R(2)\}) & \\
cl(\{T(1)\}) \arrow[ur, dash] \arrow[dr, dash] & & cl(\{R(2)\}) \arrow[ul, dash] \arrow[dl, dash] \\
& \bot &
\end{tikzcd}$$

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

smallest set that determines q. Formally, $minDet_{D,\Gamma}(Q, q)$ iff $D, \Gamma \vdash Q \twoheadrightarrow q$ and there is no $Q' \subset Q$ such that $D, \Gamma \vdash Q' \twoheadrightarrow q$. The *support of* q, denoted $supp_{D,\Gamma}(q)$, contains all sets of tuples that minimally determine q, i.e., $supp_{D,\Gamma}(q) := \{Q \in 2^{RC^{pred}} \mid minDet_{D,\Gamma}(Q, q)\}$. That is, $supp_{D,\Gamma}(q)$ 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 $supp_{D,\Gamma}(q) = \{\{q\}\}$ for all $q \in RC^{pred}$. Integrity constraints commonly used in practice, such as primary and foreign keys, satisfy this requirement, which guarantees that the information associated with a predicate query depends just on the query itself.

Disclosure orders and lattices. Bender et al. [10] 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 the 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* [10] 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 standard in lattice theory [19], 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* [10] 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 $\preceq^{\twoheadrightarrow}_{D,\Gamma}$: given Q, $Q' \in 2^{RC}$, $Q \preceq^{\twoheadrightarrow}_{D,\Gamma} Q'$ iff $D, \Gamma \vdash Q' \twoheadrightarrow Q$. Note that $Q \preceq^{\twoheadrightarrow}_{D,\Gamma} Q'$ means that Q is less informative than Q'. As shown in [10], $\preceq^{\twoheadrightarrow}_{D,\Gamma}$ 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 $R(2)$.

5.2. Security monitor

We now present our dynamic security monitor. For simplicity, we consider a single attacker, denoted by the user atk. We denote by Var_{atk} the set of variables in atk's program and by sec_0 the initial security policy.

Security lattice. Our security monitor uses the disclosure lattice to track information. To handle both queries and memory variables, we extend the database schema D with a propositional symbol MEM_x for each variable x occurring in the monitored programs. We denote by D^{ext} the extended database schema. Formally, our security lattice is the disclosure lattice $\langle \mathcal{L}, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ defined over D^{ext}, where $\sqsubseteq$ is $\preceq^{\twoheadrightarrow}_{D^{ext},\Gamma}$. Note that we use labels of the form MEM_x to abstractly represent the information initially stored in the program memories, which does not come from the database. Also observe that query determinacy is undecidable in general [34]. In §6, 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 $x \in Var$, $\Delta_0(x) = MEM_x$, (b) for all $q \in RC^{pred}$, then $\Delta_0(q) = cl(q)$, 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 along with the labels MEM_x where $x \in Var_{atk}$. Formally, $L_{\mathcal{U}}(s, u) = \top$ for any $u \notin \{atk, public\}$. For the attacker atk, $L_{\mathcal{U}}(s, atk) = cl(auth(s, atk) \cup auth(sec_0, atk) \cup \bigcup_{x \in Var_{atk}} \{MEM_x\})$. 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, $L_{\mathcal{U}}^{sec_0}(s, atk) = \bigsqcup_{v \in Val} cl(T(v)) \sqcup \bigsqcup_{x \in Var_{atk}} cl(MEM_x)$. 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 $L_{\mathcal{U}}^{sec_0}(s, u)$.

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 $cl(\{T(1), R(2)\}) \not\sqsubseteq \bigsqcup_{v \in Val} cl(T(v)) \sqcup \bigsqcup_{x \in Var_{atk}} cl(MEM_x)$.

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 be successfully executed or may 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 Appendix F. 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 $\textbf{asuser}(u', c)$, $\|x \leftarrow q\|$, $[c]$, and **set pc to** l. The $T \oplus \overline{e}$ (respectively $T \ominus \overline{e}$) query inserts (respectively deletes) the tuple $\overline{e}$ into the table T without firing triggers or throwing exceptions in case integrity constraints are violated. The $\textbf{asuser}(u', c)$ statement is used to execute the command c as the user u' (inside the session of the user u executing the $\textbf{asuser}(u', c)$ 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 [38], 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 $[c]$, which denote that the command c should be executed atomically, and **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 Appendix F.

The rule F-ASSIGN updates the monitor's state whenever there is an assignment. The rule prevents leaks using No-Sensitive Upgrade (NSU) checks [49]. 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 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 [37].

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. Furthermore, the rule keeps 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 [4], and no checks (cf. *channel context bounds* [5]) are needed.

Theorem 1, proven in Appendix H, states that our monitor is sound, i.e., it satisfies Def. 2 with $\rightsquigarrow$ as 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 $1 \leq i \leq n$, $K^{\rightsquigarrow}_{atk}(\langle M, s \rangle, C, \mathcal{S}, trace(r^{i-1})) \cap A_{atk,sec}(M, s) \subseteq K^{\rightsquigarrow}_{atk}(\langle M, s \rangle, C, \mathcal{S}, trace(r^i))$, 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): $\|y \leftarrow \texttt{SELECT } V(1)\|$; **if** y **then** $\{\|x \leftarrow T \oplus 2\|$; $\textbf{asuser}(admin, \|z \leftarrow Z \oplus 1\|)\}$ **else** $\{\|x \leftarrow T \oplus 2\|\}$.

Suppose the attacker atk executes $x \leftarrow$ `INSERT 2 INTO` $T; w \leftarrow$ `SELECT` $Z(1); \textbf{out}(atk, w)$ from a system state s_0 where the tables T and Z are empty

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

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
$$\frac{c_e = \mathit{expand}(s, x, q, u)}{\langle \Delta, x \leftarrow q, m, s\rangle \leadsto_u \langle \Delta, [c_e], m, s\rangle}$$

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

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

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

F-UPDATECONFIGURATIONOK
$$\frac{\begin{array}{c}\{v_1, \ldots, v_n\} = \mathit{vars}(q) \qquad q' = q[v_1 \mapsto [\![v_1]\!](m), \ldots, v_n \mapsto [\![v_n]\!](m)] \qquad \mathit{isCfgCmd}(q') \\ [\![q']\!](s, u) = \langle s', r, \epsilon\rangle \qquad \ell_{cmd} = \bigsqcup_{1 \le i \le n} \Delta(v_i) \qquad \ell_{cmd} \sqsubseteq \mathit{cl}(\mathit{auth}(\mathit{sec}_0, \mathit{atk})) \qquad \Delta(\mathtt{pc}_u) \sqsubseteq \mathit{cl}(\mathit{auth}(\mathit{sec}_0, \mathit{atk})) \qquad \Delta(\mathtt{pc}_u) \sqsubseteq \Delta(x)\end{array}}{\langle \Delta, \|x \leftarrow q\|, m, s\rangle \overset{\langle \mathit{public}, q'\rangle}{\rightsquigarrow}_u \langle \Delta[x \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_{cmd}], \varepsilon, m[x \mapsto r], s'\rangle}$$

Figure 4: Security monitor – selected rules.

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 initial monitor state Δ_0 such that $\Delta_0(x) = \mathit{MEM}_x$, $\Delta_0(w) = \mathit{MEM}_w$, $\Delta_0(y) = \Delta_0(z) = \top$ (since y and z do not occur in atk's program), and $\Delta_0(\mathtt{pc}_{atk}) = \bot$. The attacker's label is $L_{\mathcal{U}}(s_0, atk) = \bigsqcup_{v \in \mathit{Val}} \mathit{cl}(T(v)) \sqcup \bigsqcup_{v \in \mathit{Val}} \mathit{cl}(Z(v)) \sqcup \mathit{cl}(\{\mathit{MEM}_x, \mathit{MEM}_w\})$. The monitor would apply the rules F-EXPAND (explained above), F-SELECT, F-IFTRUE, F-UPDATEDATABASEOK, F-ASUSER (not shown), F-UPDATEDATABASEOK, 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., $\Delta'(y) = \mathit{cl}(V(1))$. The evaluation of the boolean condition y yields $\Delta' = \Delta[y \mapsto \mathit{cl}(V(1)), \mathtt{pc}_{atk} \mapsto \mathit{cl}(V(1))]$. For the subsequent database update, the monitor checks whether $\Delta'(\mathtt{pc}_{atk}) \sqsubseteq \Delta'(T(2))$, namely, whether $\mathit{cl}(V(1)) \sqsubseteq \mathit{cl}(T(2))$. Since this is not the case, the monitor stops the execution and prevents the leakage. ■

5.3. 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 `PERSON(id,name,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 $\texttt{PERSON}_{\texttt{public}}$`(id,name)` and $\texttt{PERSON}_{\texttt{secret}}$`(id,salary)`. Then, the column-level policy can be enforced using row-level policies by granting access only to $\texttt{PERSON}_{\texttt{public}}$ and not to $\texttt{PERSON}_{\texttt{secret}}$. 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 one 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 [9], [42], track column-level dependencies using the standard "low" and "high" labels.

While these two approaches are incomparable precision-wise (see Appendix A), by tracking tuple-level dependencies, we can directly support row-level policies, which are a common policy idiom from database security, and are at the basis of many fine-grained database access control models [13], [47], [36], [23]. 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. In particular, Theorem 1 guarantees that atk does not have access to sensitive information and that other users' programs do not reveal sensitive information to atk.

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

6. Disclosure lattices in practice

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

6.1. Approximating disclosure lattices

Our security monitor in §5 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 $\sqsubseteq^{abs}$.

6.2. 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 *semantics 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 $v_1 \neq v_2$.

Abstract labels. In our approximation, we track lower and upper bounds using two sets of symbolic tuples and symbols of the form MEM_x. Formally, a label ℓ is a pair $\langle S^-, S^+ \rangle$ such that S^- and S^+ are subsets of $\mathcal{P}(ST_D \cup \{MEM_x \mid x \in Var\})$. The former captures ℓ's lower bounds whereas the latter 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 functions γ^- and γ^+. For an abstract label ℓ, $\gamma^-(\ell) = cl(\bigcup_{\langle T, \varphi \rangle \in \ell|_-} \gamma(\langle T, \varphi \rangle) \cup \{MEM_x \in \ell|_- \mid x \in Var\})$ and $\gamma^+(\ell) = cl(\bigcup_{\langle T, \varphi \rangle \in \ell|_+} \gamma(\langle T, \varphi \rangle) \cup \{MEM_x \in \ell|_+ \mid x \in Var\})$. The set $\mathcal{L}^{abs}$ of all abstract labels is $\{\langle S^-, S^+ \rangle \in \mathcal{P}(ST_D \cup \{MEM_x \mid x \in Var\})^2 \mid \gamma^-(\langle S^-, S^+ \rangle) \sqsubseteq \gamma^+(\langle S^-, S^+ \rangle)\}$. We also define two distinguished elements $\top^{abs} = \langle \emptyset, ST_D \cup \{MEM_x \mid x \in Var\} \rangle$ and $\bot^{abs} = \langle \emptyset, \emptyset \rangle$.

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 (1) 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 requiring that $\gamma(\langle T, \varphi \rangle) \subseteq \gamma(\langle T, \varphi' \rangle)$), and (2) for all symbols $MEM_x \in S_1^+$, then $MEM_x \in S_2^-$. 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 the upper bounds of ℓ_1 also belongs to the lower bounds of ℓ_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 $\langle S_1^-, S_1^+ \rangle, \langle S_2^-, S_2^+ \rangle \in \mathcal{L}^{abs}$, their join $\langle S_1^-, S_1^+ \rangle \sqcup^{abs} \langle S_2^-, S_2^+ \rangle$ is $\langle S_1^- \cup S_2^-, S_1^+ \cup S_2^+ \rangle$.

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 query. Moreover, we denote $L_{\mathcal{Q}}$'s lower and upper bounds respectively by $\ell^-_{\Delta^{abs}, q}$ and $\ell^+_{\Delta^{abs}, q}$. Namely, $L_{\mathcal{Q}}^{abs}(\Delta^{abs}, q) = \langle \ell^-_{\Delta^{abs}, q}, \ell^+_{\Delta^{abs}, q} \rangle$. 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 $subs(q)$ 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 $cstrs(q) = \bigcup_{q' \in subs(q) \wedge \neg nf(q)} cstrs(q') \cup \{\langle T, \varphi\rangle \mid nf(q) \wedge q = (T(\overline{x}) \wedge \varphi)\}$. Observe that the concrete tuples represented by the symbolic tuples in $cstrs(q)$ contain those in q's support.

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 $\exists \overline{x}.\ \psi$ such that (1) $nf(\psi)$ 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 following query q: $S(1, 2) \vee \neg\exists x.\ T(1, x) \vee \exists x.\ T(2, x)$. The 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 $L^{abs}_{\mathcal{U}}$. Let sec be a security policy and $u \in UID$ be a user. The mapping $auth^{abs}(sec, u)$ is $\langle \{\langle T, \top\rangle \mid T \in auth(s, u) \cap \mathbb{T}\} \cup \{\langle T, \varphi\rangle \mid V \textit{ is a view} \wedge V \in auth(s, u) \wedge def(V) = (T(\overline{x}) \wedge \varphi) \wedge nf(def(V))\}, ST_D \cup \{MEM_x \mid x \in Var\}\rangle$. In contrast, the abstract mapping $L^{abs}_{\mathcal{U}}$ from system states and users to labels is as follows. For the attacker atk, $L^{abs}_{\mathcal{U}}(s, atk) = auth^{abs}(sec_0, atk) \sqcup^{abs} auth^{abs}(sec, atk) \sqcup^{abs} \langle \{MEM_x \mid x \in Var_{atk}\}, ST_D \cup \{MEM_x \mid x \in Var\}\rangle$, where sec_0 is the initial policy and sec is the current policy in s. For the public user $public$, $L^{abs}_{\mathcal{U}}(s, public) = L^{abs}_{\mathcal{U}}(s, atk)$. Finally, for other users u distinct from atk and $public$, $L^{abs}_{\mathcal{U}}(s, u) = \langle ST_D \cup \{MEM_x \mid x \in Var\}, ST_D \cup \{MEM_x \mid x \in Var\}\rangle$. Note that the upper bounds for $auth^{abs}(sec, u)$ and $L^{abs}_{\mathcal{U}}(s, atk)$ are always $ST_D \cup \{MEM_x \mid x \in Var\}$, 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 the monitor never uses their upper bounds.

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 $\langle \{\langle T, \top\rangle, \langle S, x_1 \neq x_2\rangle\}, ST_D \cup \{MEM_x \mid x \in Var\}\rangle$. 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 $T(\overline{v}) \in RC^{pred}$, $\Delta^{abs}_0(T(\overline{v})) = \langle \{\langle T, \bigwedge_{1 \leq i \leq |T|} x_i = v_i\rangle\}, \{\langle T, \bigwedge_{1 \leq i \leq |T|} x_i = v_i\rangle\}\rangle$, for all $x \in Var$, $\Delta^{abs}_0(x) = \langle \{MEM_x\}, \{MEM_x\}\rangle$, and for all $u \in UID$, $\Delta(\mathtt{pc}_u) = \langle \emptyset, \emptyset\rangle$.

Soundness. In Appendix I, 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.

7. Implementation and case studies

7.1. Securing SCALA programs

We now present DAISY (available at [2]), a security monitor for database-backed SCALA programs, which implements the monitor presented in §5 with the approximation from §6. DAISY enforces end-to-end security across application-database boundaries while supporting advanced database features and dynamic security policies.

Implementation. We implement DAISY via monitor inlining [17] using SCALA's macro facilities [14]. 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. Moreover, DAISY relies on the Z3 SMT solver [21] to compare symbolic tuples.

Supported fragment. To match the monitor presented in §5, DAISY only handles 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 §5 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. Additionally, these functions 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 $\Delta(\mathrm{pc}) = \bot$, and (b) they can only execute `GRANT`, `REVOKE`, and `CREATE` commands.

In DAISY, we implement a simple generalization of symbolic tuples that allows us to track dependencies across multiple tables, such as those introduced when joining several tables. In addition to symbolic tuples of the form $\langle T, \varphi\rangle$, DAISY also 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 $x_{j+\sum_{i<j}(\mathbb{T}_i|_0+\ldots+\mathbb{T}_i|_{|\mathbb{T}_i|})}$).

7.2. Case studies

To evaluate DAISY, we carried out four case studies (available at [2]): (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: (1) validate that DAISY provides the desired security guarantees, (2) confirm that our approximation is not overly restrictive, and (3) evaluate DAISY's overhead.

7.2.1. Social network. We implemented in SCALA the social network model from §2. Without the trigger, DAISY considers the program from §2 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.

7.2.2. Assignment grading system. We model a system inspired by one of URFLOW's case studies [15]. 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 a different student. We implement this policy using views and `GRANT` commands; see [2].

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." Note that 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 [15].

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) }
```

7.2.3. 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 [2] for further details.

7.2.4. 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 `review(paperID, revID, 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 $\texttt{review}^{\mathrm{A}}_u = \{p, r, d \mid \texttt{review}(p, r, d) \wedge \texttt{author}(p, u)\}\}$. 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 $\texttt{review}^{\mathrm{R}}_u = \{p, r, d \mid \texttt{review}(p, r, d) \wedge \exists c, t.\ (\texttt{paper}(p, c, t) \wedge \texttt{reviewer}(c, u))\}$. We can now define the permissions. Whenever a user u acts as author, he can read $\texttt{review}^{\mathrm{A}}_u$. In contrast, when a user u acts as reviewer, he can read $\texttt{review}^{\mathrm{R}}_u$. 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 in 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 relies on 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 the information stored in the variable *data* 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.

7.2.5. 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 unmonitored execution of the same code, which we believe is acceptable for a proof-of-concept implementation.

8. Related work

IFC for database-backed applications. We compare our work with existing IFC solutions for database-backed applications [27], [16], [30], [43], [18], [15], [9], [48] w.r.t. 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 the aforementioned criteria.

SIF [16] enforces IFC policies for Java web applications, whereas Li and Zdancewic [30] 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 [43], a system supporting decentralized IFC across databases and applications. Its *Query by Label* model extends the work on multi-level secure (MLS) databases [32] and provides abstractions for dealing with expressive IFC policies. It supports complex database features and policies. Similarly to other MLS approaches, it relies on polyinstantiation [29], which is not supported by the SQL standard and requires ad-hoc extensions [22], [41]. 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 [9], SELINKS [18], [44], and SELINQ [42] secure applications that interact with databases through language-integrated queries. In contrast to DAISY, these works consider simpler database models and ignore constructs like triggers and integrity constraints. Moreover, JSLINQ and SELINQ only support column-level policies, while SELINKS can also support row-level policies. However, none of them support row-level policies where privileges can be granted and revoked as we do. Lourenço and

	DATABASE FEATURES					SECURITY POLICIES		Soundness proof
	`INSERT` - `DELETE`	Dynamic policies	Triggers	Integrity constraints	Views	Column level	Row level	
SIF [16]	✓	✓^{1}				✓		✓
Li et al. [30]		✓^{1}				✓		
IFDB [43]	✓	✓^{1}	✓	✓	✓	✓	✓	
JSLINQ [9]		✓^{1}				✓		✓
SELINKS [18]	✓	✓^{1}				✓	✓	✓
SELINQ [42]						✓		✓
Lourenço et al. [31]	✓					✓	✓	✓
URFLOW [15]	✓					✓	✓	
LWEB [35]	✓	✓^{1}				✓	✓	✓
JACQUELINE [48]	✓	✓				✓	✓	✓
DAISY	✓	✓	✓	✓	✓	✓^{2}	✓	✓

^{1}Only declassification ^{2}Only static column-level policies

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

Caires [31] 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, therefore they ignore the challenges posed by advanced database features and dynamic policies.

URFLOW [15] 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 [35] 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 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 [48] presents an IFC approach that secures database-backed applications using faceted execution [7]. 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 [5], [8], [12]. 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 [5] propose security conditions against all attackers. We conjecture that our security monitor also enforces security against all attackers. Hicks et al. [26] propose *non-interference between updates* which ensures non-interference between policy changes, while ignoring information leaks across such changes. We refer the reader to Broberg et al. [12] for a survey of dynamic policies.

Label models. The *universal lattice* [28] allows one to express 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 [47], [36], [23], [11], [10]. Guarnieri et al. [24] 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. [11], [10] 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 [33] 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 [22], [32], 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 *tran-*

quility principle [40]) and rely on poly-instantiation [29].

9. 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.

References

[1] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of databases*. Addison-Wesley Reading, 1995, vol. 8.

[2] Anonymized, "DAISY: DAtabase and Information-flow SecuritY," https://sites.google.com/site/databaseifc/, 2018.

[3] ——, "Information-Flow Control for Database-backed Applications – Technical Report," https://sites.google.com/site/databaseifc/, 2018.

[4] O. Arden, J. Liu, and A. C. Myers, "Flow-limited authorization," in *CSF*, 2015.

[5] A. Askarov and S. Chong, "Learning is change in knowledge: Knowledge-based security for dynamic policies," in *CSF*, 2012.

[6] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, "Termination-insensitive noninterference leaks more than just a bit," in *ESORICS*, 2008.

[7] T. H. Austin and C. Flanagan, "Multiple facets for dynamic information flow," in *ACM Sigplan Notices*, vol. 47, no. 1. ACM, 2012, pp. 165–178.

[8] M. Balliu, "A logic for information flow analysis of distributed programs," in *NordSec*, 2013.

[9] M. Balliu, B. Liebe, D. Schoepe, and A. Sabelfeld, "Jslinq: Building secure applications across tiers," in *CODASPY*, 2016.

[10] G. Bender, L. Kot, and J. Gehrke, "Explainable security for relational databases," in *SIGMOD*, 2014.

[11] G. Bender, L. Kot, J. Gehrke, and C. Koch, "Fine-grained disclosure control for app ecosystems," in *SIGMOD*, 2013.

[12] N. Broberg, B. van Delft, and D. Sands, "The anatomy and facets of dynamic policies," in *CSF*, 2015.

[13] K. Browder and M. Davidson, "The virtual private database in Oracle9iR2," *Oracle Technical White Paper, Oracle Corporation*, vol. 500, 2002.

[14] E. Burmako, "Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming," in *SCALA@ECOOP*, 2013.

[15] A. Chlipala, "Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications." in *OSDI*, 2010.

[16] S. Chong, K. Vikram, and A. C. Myers, "SIF: Enforcing Confidentiality and Integrity in Web Applications," in *USENIX Security*, 2007.

[17] A. Chudnov and D. A. Naumann, "Information flow monitor inlining," in *CSF*, 2010.

[18] B. J. Corcoran, N. Swamy, and M. W. Hicks, "Cross-tier, label-based security enforcement for web applications." in *SIGMOD*, 2009.

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

[20] B. Davis and H. Chen, "DBTaint: cross-application information flow tracking via databases," in *WebApps*, 2010.

[21] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS'08*, 2008.

[22] D. E. Denning and T. F. Lunt, "A multilevel relational data model," in *S&P*, 1987.

[23] M. Guarnieri and D. Basin, "Optimal security-aware query processing," in *VLDB*, 2014.

[24] M. Guarnieri, S. Marinovic, and D. Basin, "Strong and provably secure database access control," in *EuroS&P*, 2016.

[25] D. Hedin and A. Sabelfeld, "A perspective on information-flow control," in *Software Safety and Security - Tools for Analysis and Verification*, 2012, pp. 319–347.

[26] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic, "Dynamic updating of information-flow policies," in *FCS*, 2005.

[27] 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.

[28] S. Hunt and D. Sands, "On flow-sensitive security types," in *POPL*, 2006.

[29] S. Jajodia and R. Sandhu, "Polyinstantiation integrity in multilevel relations," in *S&P*, 1990.

[30] P. Li and S. Zdancewic, "Practical information flow control in web-based information systems," in *CSF*, 2005.

[31] L. Lourenço and L. Caires, "Dependent information flow types," in *POPL'15*, 2015.

[32] 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.

[33] A. Mehta, E. Elnikety, K. Harvey, D. Garg, and P. Druschel, "Qapla: Policy compliance for database-backed systems," in *USENIX Security*, 2017.

[34] A. Nash, L. Segoufin, and V. Vianu, "Views and queries: Determinacy and rewriting," *TODS*, vol. 35, no. 3, p. 21, 2010.

[35] J. Parker, N. Vazou, and M. Hicks, "LWeb: Information flow security for multi-tier web applications," in *POPL'19*, 2019.

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

[37] A. Russo and A. Sabelfeld, "Dynamic vs. static flow-sensitive security analysis," in *CSF*, 2010.

[38] A. Sabelfeld and D. Sands, "Probabilistic noninterference for multi-threaded programs," in *CSFW*, 2000.

[39] P. Samarati, "Recursive revoke," in *Encyclopedia of Cryptography and Security*. Springer, 2011, pp. 1035–1037.

[40] P. Samarati and S. Capitani de Vimercati, "Access Control: Policies, Models, and Mechanisms," *Springer LNCS*, vol. 2171, 2001.

[41] R. Sandhu and F. Chen, "The multilevel relational (MLR) data model," *TISSEC*, vol. 1, no. 1, 1998.

[42] D. Schoepe, D. Hedin, and A. Sabelfeld, "Selinq: Tracking information across application-database boundaries," in *ICFP*, 2014.

[43] D. Schultz and B. Liskov, "IFDB: decentralized information flow control for databases," in *EuroSys*, 2013.

[44] N. Swamy, B. J. Corcoran, and M. Hicks, "Fable: A language for enforcing user-defined security policies," in *S&P'08*, 2008.

[45] B. van Delft, S. Hunt, and D. Sands, "Very static enforcement of dynamic policies," in *POST*, 2015.

[46] S. D. C. d. Vimercati and G. Livraga, "Sql access control model," in *Encyclopedia of Cryptography and Security*. Springer, 2011, pp. 1248–1251.

[47] 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.

[48] 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.

[49] S. A. Zdancewic, "Programming languages for information security," Ph.D. dissertation, Cornell University, Ithaca, NY, USA, 2002.

System and runtime states (§3)	
D	database schema
Γ	set of integrity constraints
$M = \langle D, \Gamma \rangle$	system configuration
db	database state
U	set of users
sec	security policy
T	set of triggers
V	set of views
$s = \langle db, U, sec, T, V \rangle$	system state
ctx	database context
$\langle s, ctx \rangle \in \Omega_M$	runtime state
Security monitor (§5)	
q	query
Q	set of queries
$T(\overline{v}) \in RC^{pred}$	predicate query
$supp(q)$	support of query q
$cl(q)$	closure of query q
$\mathcal{L}$	disclosure lattice
$D, \Gamma \vdash Q \twoheadrightarrow Q'$	Q determines Q' given D, Γ
$Q' \preceq^{\twoheadrightarrow}_{D,\Gamma} Q$	disclosure order (iff $D, \Gamma \vdash Q \twoheadrightarrow Q'$)
$\sqsubseteq$	order in the lattice
$\sqcup$	join operator in the lattice
Δ_0, Δ	(initial) monitor state
$\mathtt{pc}_u$	u's program context
$L_{\mathcal{Q}}(\Delta, q)$	mapping queries to labels
$L_{\mathcal{U}}(s, u)$	mapping users to labels
$\rightsquigarrow_u$	monitor's local semantics
$\rightsquigarrow$	monitor's global semantics

WHILESQL (§3)	
Var, Val	variables and values
$c \in Com$	WHILESQL program
$m \in Mem$	memories
$\langle c, m, \langle s, ctx \rangle \rangle \in Conf$	local configuration
$Com_{UID} = UID \times Com$	programs and executing users
$Mem_{UID} = UID \times Mem$	memories and executing users
$C \in Com^*_{UID}$	sequence of programs
$M \in Mem^*_{UID}$	sequence of memories
$\mathcal{S}$	scheduler
$\langle C, M, \langle s, ctx \rangle, \mathcal{S} \rangle \in GlConf$	global configurations
$\langle M, s \rangle$	global state
$\rightarrow_u$	local semantics
$\rightarrow$	global semantics
r	run
$\langle u, o \rangle$	observation
$trace(r)$	r's observations
Security condition (§4)	
$K_u(\langle M_0, s_0 \rangle, C, \mathcal{S}, \tau)$	attacker's knowledge
$A_{u,sec}(M_0, s_0)$	allowed knowledge given sec
Approximation (§6)	
$\langle T, \varphi \rangle$	symbolic tuple
$\langle S^-, S^+ \rangle \in \mathcal{L}^{abs}$	abstract label
$\sqsubseteq^{abs}$	order over abstract labels
$\sqcup^{abs}$	join operator over abstract labels
Δ^{abs}	abstract monitor state
$L^{abs}_{\mathcal{Q}}(\Delta^{abs}, q)$	mapping queries to abstract labels
$L^{abs}_{\mathcal{U}}(s, u)$	mapping users to abstract labels
$cstrs(q)$	symbolic tuples extracted from query q

Figure 6: Table of notation

Appendix A. Tracking dependencies between tuples and columns

By leveraging disclosure lattices, our monitor tracks dependencies at the tuple-level. In contrast, other approaches, such as [9], [42], track dependencies at the column-level. These two approaches are incomparable precision-wise. We now illustrate the advantages and drawbacks of these two approaches.

Tracking dependencies at the tuple-level allows one to differentiate between tuples in the same table when checking for possible leaks. For instance, consider the following snippet, where $high$ is a variable containing information that atk is not authorized to read:

$$\begin{array}{l} x \leftarrow \texttt{INSERT}\ \langle 1, high\rangle\ \texttt{INTO}\ T \\ y \leftarrow \texttt{SELECT}\ T(2,2) \\ \textbf{out}(atk, y) \end{array}$$

The above snippet is always secure, since the SELECT query's result does not depend on $high$. An IFC mechanism that tracks dependencies at the tuple-level, such as our monitor, can determine this, since it can differentiate between the tuples $T(2,2)$ and $T(1, high)$. In contrast, mechanisms that track dependencies at column-level, such as [9], [42], cannot distinguish between these two tuples, and thereby reject the above snippet as insecure.

Column-level dependency tracking allows one to track whether sensitive information has been inserted in specific columns. Consider the following snippet:

$$\begin{array}{l} x \leftarrow \texttt{INSERT}\ \langle 1, high\rangle\ \texttt{INTO}\ T \\ y \leftarrow \texttt{SELECT}\ \exists z.\ T(1,z) \\ \textbf{out}(atk, y) \end{array}$$

Again, this snippet is secure. A mechanism that tracks dependencies at the column-level can determine that the SELECT query's result does not depend on the value z, which belongs to the column containing sensitive information. Tuple-level dependency tracking, however, is not sufficiently precise to differentiate between different columns. Hence, mechanisms tracking dependency at the tuple-level, such as our monitor, reject the above snippet as insecure.

To summarize, tuple-level and column-level dependency tracking are orthogonal and incomparable. The former allows one to track sensitive information across tables (abstracting away from columns), whereas the latter allows the tracking of sensitive information in specific columns (but across tuples). We believe that a closer integration of tuple-level and column-level dependency tracking may lead to improvements in IFC precision. We leave this as future work.

Appendix B. Progress-sensitivity

Theorem 1 states that our monitor from §5 is sound for our progress-insensitive security condition. A progress-sensitive variant of our security condition can be obtained by replacing the knowledge $K_u(\langle M_0, s_0\rangle, C, \mathcal{S}, \tau)$ with the progress-sensitive knowledge $PK_u(\langle M_0, s_0\rangle, C, \mathcal{S}, \tau) = \{\langle M, s\rangle \mid s \approx_u s_0 \wedge M \approx_u M_0 \wedge \exists ctx', \tau', C', M', s', \mathcal{S}'.\ \langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau'}{}^* \langle C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle \wedge \tau\lceil_{db(u)} = \tau'\lceil_{db(u)}\}$ in Def. 2.

To achieve progress-sensitive security, it suffices to ensure that branching statements are executed only in case the level of the statement's condition is "low", i.e., one can simply add the check $\Delta'(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, atk)$ to the F-IfTrue, F-IfFalse, F-WhileTrue, and F-WhileFalse rules.

Appendix C. Relaxing NSU checks

In our monitor from §5, NSU checks of the form $\Delta(\mathtt{pc}_u) \sqsubseteq \Delta(x)$ can be replaced with checks of the form $\Delta(\mathtt{pc}_u) \sqsubseteq cl(auth(sec_0, atk)) \vee \Delta(\mathtt{pc}_u) \sqsubseteq \Delta(x)$, where sec_0 is the initial security policy, without affecting the monitor's soundness. This check exploits the initial policy sec_0 to determine which labels can be considered as permanently "low", no matter how the policy changes at runtime. This improves our monitor's permissiveness. In our experiments in §7, we did not face precision issues. However, precision issues, e.g., due to NSU checks, can be solved by code annotations.

Appendix D. Social Networking example

Here, we encode the programs from §2 in the Scala's fragment supported by Daisy.

The first program P_1 retrieves all the friends of the user $user$ that is executing the program. For each of $user$'s friends u_1, P_1 retrieves all reviews of books that both $user$ and u_1 have read and forwards them to mutual friends of $user$ and u_1. We assume that SQL injection vulnerabilities, such as through the string interpolation `s"..$user.."`, are prevented by a separate mechanism.

```
val friends = select(s"{ x | EXISTS y. ((friends(y, x) AND y = '$user')) }").map(_.head)
val reviews = select(s"{ b | EXISTS u,r. ((reviews(b, u, r) AND u = '$user')) }").map(_.head)
friends.foreach { friend =>
  val friendReviews = select(s"{ b, r | EXISTS u. ((reviews(b, u, r) AND u = '$friend')) }")
  val friendFriends = select(s"{ uu | EXISTS u. ((friends(u, uu) AND u = '$friend' )) }").map(_.head)
  (friendFriends intersect friends).foreach { u2 =>
    friendReviews.foreach { r =>
      if (reviews contains r.head) { outputTo(u2, r) } } } }
```

The program is secure, since the information that flows to other users through the output statements comply with the security policy, and Daisy correctly accepts it as secure.

The second program P_2 models the behavior of an attacker that (1) becomes friends with another user (denoted by $target$) and (2) discloses publicly the $target$'s reviews. The program P_2, executed by atk, is as follows:

```
asAdmin { addFriend("attacker", "target") }
val reviews = select("{ b, r | EXISTS u. ((reviews(b, u, r) AND u = 'target')) }")
reviews.foreach { review => outputTo("public", review) }
```

This program violates our security policy since the attacker atk leaks the reviews of $target$ publicly, and Daisy correctly blocks the program's execution. This example demonstrates that the presented approach can track the interplay of database features and information flow tracking at application level.

Basic Types

(Table Ids)	$T \in \mathbb{T}$	*(Variables)*	$x \in \mathit{Var}$	*(Trigger Ids)*	$tr \in \mathbb{TR}$
(View Ids)	$V \in \mathbb{V}$	*(Values)*	$n \in \mathit{Val}$	*(Formulae)*	$\varphi \in \mathit{RC}$
(Relation Ids)	$R \in \mathbb{T} \cup \mathbb{V}$	*(User identifiers)*	$u \in \mathcal{U}$	*(Error Messages)*	$em \in \mathit{EM}$

Syntax

(User Context)	uc	$:=$ `OWNER` $\mid$ `INVOKER`
(Privileges)	p	$:=$ `SELECT ON` R $\mid$ `INSERT ON` T $\mid$ `DELETE ON` T $\mid$ `CREATE VIEW` $\mid$ `CREATE TRIGGER ON` T
(Actions)	a	$:=$ `INSERT` $e_1, \ldots, e_n$ `INTO` T $\mid$ `DELETE` $e_1, \ldots, e_n$ `FROM` T
		$\mid$ `GRANT` p `TO` u $\mid$ `GRANT` p `TO` u `WITH GRANT OPTION` $\mid$ `REVOKE` p `FROM` u
(SQL commands)	q	$:=$ a $\mid$ `SELECT` φ $\mid$ `ADD USER` u $\mid$ `CREATE VIEW` V : `SELECT` φ
		$\mid$ `CREATE TRIGGER` tr `ON` T `AFTER (INS`$\mid$`DEL) IF` φ `DO` a
(Expressions)	e	$:= n \mid x \mid \neg e \mid e_1 \oplus e_2$
(Statements)	c	$:= \varepsilon \mid x \leftarrow q \mid x := e \mid \mathbf{out}(u, e) \mid \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \mid \mathbf{while}\ e\ \mathbf{do}\ c \mid c_1\ ;\ c_2$

Figure 7: WHILESQL's syntax.

Appendix E. WHILESQL

WHILESQL is a simple language that captures the main features of both programming languages extended with querying constructs and procedural extensions of the SQL standard, such as Oracle's PL/SQL or Microsoft TRANSACT-SQL. At the same time, it simplifies some subtle aspects of their semantics, while still capturing the main security-critical features.

E.1. Syntax

Figure 7 depicts WHILESQL's syntax. Let $\mathbb{T}$, $\mathbb{V}$, and $\mathbb{TR}$ be three countably infinite sets representing table identifiers, view identifiers, and trigger identifiers. Furthermore, let Var and Val be countably infinite sets of variables and values. We assume that all these sets are pairwise disjoint.

As shown in Figure 7, a WHILESQL program is an imperative program extended with querying capabilities, i.e., statements of the form $x \leftarrow q$. A statement $x \leftarrow q$ executes the SQL command q and assigns its result to the variable x. WHILESQL supports SQL's core features, such as `SELECT`, `INSERT`, `DELETE`, `GRANT`, and `REVOKE` commands, as well as advanced database features such as triggers and views. Additionally, WHILESQL programs support assignments and standard control flow statements. WHILESQL also supports $\mathbf{out}(u, e)$ statements to output the value of the expression e to the user u. For simplicity, we assume that all expressions are well-typed and all SQL statements refer either to tables in the database schema or to previously created views.

WHILESQL builds on top of the database operational semantics developed by Guarnieri et al. [24]. Hence, it supports the same fragment of SQL supported by Guarnieri et al.'s database semantics. We now recall the restrictions and simplifications inherited from our operational semantics. For `SELECT` commands, instead of using SQL, we rely on the relational calculus. Moreover, we support only `INSERT` and `DELETE` commands that explicitly identify the tuple to be inserted or deleted. Finally, we support only triggers that are executed in response to `INSERT` and `DELETE` commands. We assume that a trigger's body has the form `IF` φ `DO` a, where φ is a boolean query and a is an `INSERT` or `DELETE` command.

Each SQL command either returns the query result or an error message $em \in \mathit{EM}$. Error messages indicate whether queries (or triggers) violate security constraints, like a query that is not allowed by the current security policy, or integrity constraints, such as an `INSERT` statement that violates a primary key constraint. Note that error messages are values, i.e., $\mathit{EM} \subseteq \mathit{Val}$.

E.2. Local Semantics

Here, we define the semantics of a WHILESQL program executed in isolation. A WHILESQL program is defined with respect to a *system configuration* $M = \langle D, \Gamma \rangle$, where $D = \langle \Sigma, \mathbf{dom} \rangle$ is a database schema and Γ is a set of integrity constraints. We assume that $\mathbf{dom} \subseteq \mathit{Val}$ and that only values in $\mathbf{dom}$ are used to construct queries. For simplicity, we fix a configuration $M = \langle D, \Gamma \rangle$ for the rest of the section.

Databases. WHILESQL reuses the database model and the notion of system and runtime states from [24]. Here, we recall only the main concepts that we use in the rest of the chapter. We refer the reader to [24] for more details on security policies and our database model.

A *security policy* is a finite set of `GRANT` statements. Given a policy sec and a user u, we denote by $\mathit{auth}(sec, u)$ the set of all tables and views with the owner's privileges that u is authorized to read according to the `GRANT` statements in sec. A *system state* is a tuple $\langle db, U, sec, T, V \rangle$ where db is a database state, $U \subset \mathit{UID}$ is a finite set of users, T is a finite set of triggers, V is a finite set of views, and sec is a security policy. Note that we lift auth from policies to system states, i.e., $\mathit{auth}(\langle db, U, sec, T, V \rangle, u) = \mathit{auth}(sec, u)$. A *context* ctx describes the database's history,

E-SKIP
$$\frac{}{\langle \textbf{skip}, m, s\rangle \rightarrow_u \langle \varepsilon, m, s\rangle}$$

E-ASSIGN
$$\frac{}{\langle x := e, m, s\rangle \rightarrow_u \langle \varepsilon, m[x \mapsto [\![e]\!](m)], s\rangle}$$

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

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

E-OUT
$$\frac{}{\langle \textbf{out}(u', e), m, s\rangle \xrightarrow{\langle u', [\![e]\!](m)\rangle}_u \langle \varepsilon, m, s\rangle}$$

E-IFTRUE
$$\frac{[\![e]\!](m) = \textbf{tt}}{\langle \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, m, s\rangle \rightarrow_u \langle c_1, m, s\rangle}$$

E-IFFALSE
$$\frac{[\![e]\!](m) = \textbf{ff}}{\langle \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, m, s\rangle \rightarrow_u \langle c_2, m, s\rangle}$$

E-WHILETRUE
$$\frac{[\![e]\!](m) = \textbf{tt}}{\langle \textbf{while } e \textbf{ do } c, m, s\rangle \rightarrow_u \langle c\ ;\ \textbf{while } e \textbf{ do } c, m, s\rangle}$$

E-WHILEFALSE
$$\frac{[\![e]\!](m) = \textbf{ff}}{\langle \textbf{while } e \textbf{ do } c, m, s\rangle \rightarrow_u \langle \varepsilon, m, s\rangle}$$

E-SEQ
$$\frac{\langle c_1, m, s\rangle \xrightarrow{\tau}_u \langle c_1', m', s'\rangle}{\langle c_1\ ;\ c_2, m, s\rangle \xrightarrow{\tau}_u \langle c_1'\ ;\ c_2, m', s'\rangle}$$

E-SEQEMPTY
$$\frac{}{\langle \varepsilon\ ;\ c, m, s\rangle \rightarrow_u \langle c, m, s\rangle}$$

Figure 8: WHILESQL's local operational semantics.

the scheduled triggers that must be executed, and how to modify the database's state in case a roll-back occurs. A *runtime state* is a tuple $\langle s, \mathit{ctx}\rangle$ where s is a system state and ctx is a context. The set of all runtime states is denoted by Ω_M and we denote by ϵ the empty context. In the following, we use s to refer to both system states and runtime states whenever this is clear from the context, and we use the notation $\langle s, \mathit{ctx}\rangle$ otherwise.

Memories and Configurations. A *memory* $m \in \mathit{Mem}$ is a partial function mapping variables to values, i.e. $\mathit{Mem} : \mathit{Var} \to \mathit{Val}$. A *local configuration* $\langle c, m, \langle s, \mathit{ctx}\rangle\rangle$ consists of a command $c \in \mathit{Com}$, a memory $m \in \mathit{Mem}$, and a runtime state $\langle s, \mathit{ctx}\rangle \in \Omega_M$. A local configuration is *initial* iff its context ctx is ϵ and the database state s is an initial database state as defined in [24]. We denote by Conf the set of all configurations.

Users. Let *UID* be a countably infinite set representing all user identifiers. In addition to users in *UID*, we add a designated user public that can observe only public events (i.e., changes to the database configuration).

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

Semantics. Given a user $u \in \mathit{UID}$, the relation $\rightarrow_u \subseteq (\mathit{Com} \times \mathit{Mem} \times \Omega_M) \times \mathit{Obs} \times (\mathit{Com} \times \mathit{Mem} \times \Omega_M)$ formalizes the small-step local operational semantics of WHILESQL 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 $\rightarrow_u$. 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 associated to a run r, denoted by $\mathit{trace}(r)$, is obtained by concatenating all observations in the run.

As noted above, the operational semantics of SQL statements relies on the operational semantics given in [24]. We use the function $[\![q]\!](\langle s, \mathit{ctx}\rangle, u)$ (defined in Appendix E.4) to connect WHILESQL's operational semantics with the database operational semantics from [24]. The function $[\![q]\!](\langle s, \mathit{ctx}\rangle, u)$ takes as input an SQL command q, a runtime state $\langle s, \mathit{ctx}\rangle \in \Omega_M$, and the user $u \in \mathit{UID}$ executing the command, and it returns a tuple $\langle\langle s, \mathit{ctx}\rangle', r, \mathit{em}\rangle$, where $\langle s', \mathit{ctx}'\rangle \in \Omega_M$ is the new runtime state, r is q's result, and em is an error message. Note that em is ϵ in case executing the command does not generate error messages. 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 query or expression evaluation.

Figure 8 depicts the rules defining WHILESQL's local semantics. Most of the rules are standard. The only non-standard rules are E-QUERYOK and E-QUERYEX, which regulate the execution of SQL commands. The rule E-QUERYOK models the successful execution of SQL commands. It first replaces the free variables in the query with their actual values. Afterwards, it executes the query and it stores the query result in the memory. The rule relies on the function $\mathit{obs}(q)$ to produce observations associated with the successful execution of `GRANT`, `REVOKE`, and `CREATE` commands. Formally, $\mathit{obs}(q)$ is $\langle \mathit{public}, q\rangle$ in case q is a `GRANT`, `REVOKE`, or `CREATE` command, and $\mathit{obs}(q)$ is the

M-EVAL-STEP

$$\frac{\begin{array}{c}\forall i \in \{1,\dots,|C|\}, u' \in \mathit{UID}.\ C|_i \neq \langle u', \varepsilon\rangle \quad n = 1 + (n' \ \mathbf{mod}\ |C|) \quad C|_n = \langle u, c\rangle \quad M|_n = \langle u, m\rangle \\ |C| = |M| \quad \langle c, m, s\rangle \xrightarrow{\tau}_u \langle c', m', s'\rangle \quad C' = C|_1 \cdot \ldots \cdot C|_{n-1} \cdot \langle u, c'\rangle \cdot C|_{n+1} \cdot \ldots \cdot C|_{|C|} \\ M' = M|_1 \cdot \ldots \cdot M|_{n-1} \cdot \langle u, m'\rangle \cdot M|_{n+1} \cdot \ldots \cdot M|_{|C|}\end{array}}{\langle C, M, s, n' \cdot \mathcal{S}\rangle \xrightarrow{\tau} \langle C', M', s', \mathcal{S}\rangle}$$

M-EVAL-END

$$\frac{\begin{array}{c}1 \leq n \leq |C| \quad \forall n' < n, u' \in \mathit{UID}.\ C|_{n'} \neq \langle u', \varepsilon\rangle \quad C|_n = \langle u, \varepsilon\rangle \quad |C| = |M| \\ C' = C|_1 \cdot \ldots \cdot C|_{n-1} \cdot C|_{n+1} \cdot \ldots \cdot C|_{|C|} \qquad M' = M|_1 \cdot \ldots \cdot M|_{n-1} \cdot M|_{n+1} \cdot \ldots \cdot M|_{|C|}\end{array}}{\langle C, M, s, \mathcal{S}\rangle \xrightarrow{\tau} \langle C', M', s, \mathcal{S}\rangle}$$

Figure 9: WHILESQL's global operational semantics

empty trace ε otherwise. The rule E-QUERYEX, instead, models the failed execution of a query. The rule executes the query, retrieves the error message, and stores it in the memory.

E.3. Global Semantics

To model realistic scenarios, where attackers and honest users each run their own programs which may access a common database, we assume that programs do not share memory, whereas the database is shared. We now present a global semantics capturing the parallel execution of WHILESQL programs.

Schedulers. We model a *scheduler* as an infinite sequence of natural numbers $\mathcal{S} \in \mathbb{N}^\omega$. In the global semantics, we use the scheduler to determine which program has to be executed at each point in the execution.

Global Configurations. We denote the set of commands together with the executing user by $\mathit{Com}_{\mathit{UID}} = \mathit{UID} \times \mathit{Com}$ and the set of pairs of users and memories as $\mathit{Mem}_{\mathit{UID}} = \mathit{UID} \times \mathit{Mem}$. 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 $\langle C, M, \langle s, \mathit{ctx}\rangle, \mathcal{S}\rangle \in \mathit{GlConf}$, where $C \in \mathit{Com}^*_{\mathit{UID}}$ is a sequence of WHILESQL programs with the executing users, $M \in \mathit{Mem}^*_{\mathit{UID}}$ is a sequence of memories (one per program in C), $\langle s, \mathit{ctx}\rangle \in \Omega_M$ 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 $\langle C, M, s, \mathcal{S}\rangle$ such that $|C| = |M|$ and for all $1 \leq i \leq |C|$, $C|_i = \langle u, c\rangle$, $M|_i = \langle u', m\rangle$, and $u = u'$. Furthermore, a *global state* is a pair $\langle M, s\rangle$, where $M \in \mathit{Mem}^*_{\mathit{UID}}$ and s is a system state.

Semantics. The relation $\rightarrow \subseteq \mathit{GlConf} \times \mathit{Obs} \times \mathit{GlConf}$, shown in Figure 9, formalizes the global operational semantics of a database system that runs multiple WHILESQL programs in parallel. Given a global configuration $\langle C, M, s, \mathcal{S}\rangle$, the global operational semantics uses the scheduler $\mathcal{S}$ to select which of the programs in C to execute. This is done by extracting the first number n' from the scheduler $\mathcal{S}$ and identifying the associated program $\langle u, c\rangle$ and memory $\langle u, m\rangle$ in C and M respectively. The rule M-EVAL-STEP identifies the WHILESQL program that should be executed according to the scheduler, it executes one step of the local semantics, and it updates the global state accordingly. The rule M-EVAL-END, instead, removes the terminated programs from the global configuration. Given a run r, we denote by $\mathit{conf}(r)$ the global configuration in the last state in the run. Furthermore, $\mathit{trace}(r)$ denotes the trace associated with the run r, and $\mathit{db}(r)$ denotes the database state in the global configuration $\mathit{conf}(r)$.

E.4. From WHILESQL to the database operational semantics of [24]

The WHILESQL's operational semantics builds on top of the database operational semantics formalized in [24]. In particular, in the WHILESQL semantics, the execution of the database commands is delegated to the function $[\![q]\!](s, u)$, which takes as input an SQL statement q, a runtime state $s \in \Omega_M$, and the user $u \in \mathit{UID}$ executing the command, and it returns a tuple $\langle s', r, \mathit{em}\rangle$, where $s' \in \Omega_M$ is the new runtime state, r is q's result, em is an error message. Note that $\mathit{em} = \epsilon$ in case there is no error message. The function $[\![q]\!](s, u)$ is defined in Figures 10–12 and it relies on the transition relation $\rightarrow_f$ from [24], where f is a PDP. In the following, we instantiate f to be the PDP f_{int} developed in [24]. This PDP ensures the integrity of the database, e.g., by avoiding unauthorized changes, but it does not provide confidentiality guarantees. Furthermore, we re-use various functions from [24], e.g., we reuse the functions res, Ex, and secEx to extract the outcomes of the command's execution from a runtime state, and we lift the auth function from system states to runtime states, i.e., $\mathit{auth}(\langle s, \mathit{ctx}\rangle, u)$ is simply $\mathit{auth}(s, u)$.

We remark that $[\![q]\!](\langle s, \mathit{ctx}\rangle, u)$ guarantees that the trigger transaction, which has been defined in [24], in the resulting runtime state $\langle s', \mathit{ctx}'\rangle$ is always ϵ. This, combined with the fact that the context for initial states is ϵ, ensures the correct execution of queries. As a consequence of this, $[\![q]\!](\langle s, \mathit{ctx}\rangle, u) = [\![q]\!](\langle s, \mathit{ctx}'\rangle, u)$ for any two contexts $\mathit{ctx}, \mathit{ctx}'$ such that the trigger transaction is ϵ.

For SELECT queries, $[\![q]\!](s, u)$ is defined only for boolean queries because the operational semantics in [24] supports only boolean queries. We refer the reader to [24] for a discussion on how to handle non-boolean queries.

$$[\![\texttt{SELECT}\ \phi]\!](s,u) = \begin{cases} \langle s', res(s'), \epsilon\rangle & \text{if } s \xrightarrow{\langle u,\texttt{SELECT},\phi\rangle}_f s' \wedge \neg secEx(s') \\ \langle s', \dagger, \langle \mathbf{SecEx}, \emptyset\rangle\rangle & \text{if } s \xrightarrow{\langle u,\texttt{SELECT},\phi\rangle}_f s' \wedge secEx(s') \end{cases}$$

$$[\![\texttt{CREATE}\ obj]\!](s,u) = \begin{cases} \langle s', res(s'), \epsilon\rangle & \text{if } s \xrightarrow{\langle u,\texttt{CREATE},obj\rangle}_f s' \wedge \neg secEx(s') \\ \langle s', \dagger, \langle \mathbf{SecEx}, \emptyset\rangle, \epsilon\rangle & \text{if } s \xrightarrow{\langle u,\texttt{CREATE},obj\rangle}_f s' \wedge secEx(s') \end{cases}$$

$$[\![\texttt{GRANT}\ p\ \texttt{TO}\ u']\!](s,u) = \begin{cases} \langle s', res(s'), \epsilon\rangle & \text{if } s \xrightarrow{\langle \oplus,u,p,u'\rangle}_f s' \wedge \neg secEx(s') \\ \langle s', \dagger, \langle \mathbf{SecEx}, \emptyset\rangle, \epsilon\rangle & \text{if } s \xrightarrow{\langle \oplus,u,p,u'\rangle}_f s' \wedge secEx(s') \end{cases}$$

$$[\![\texttt{GRANT}^*\ p\ \texttt{TO}\ u']\!](s,u) = \begin{cases} \langle s', res(s'), \epsilon\rangle & \text{if } s \xrightarrow{\langle \oplus^*,u,p,u'\rangle}_f s' \wedge \neg secEx(s') \\ \langle s', \dagger, \langle \mathbf{SecEx}, \emptyset\rangle, \epsilon\rangle & \text{if } s \xrightarrow{\langle \oplus^*,u,p,u'\rangle}_f s' \wedge secEx(s') \end{cases}$$

$$[\![\texttt{REVOKE}\ p\ \texttt{FROM}\ u']\!](s,u) = \begin{cases} \langle s', res(s'), \epsilon\rangle & \text{if } s \xrightarrow{\langle \ominus,u,p,u'\rangle}_f s' \wedge \neg secEx(s') \\ \langle s', \dagger, \langle \mathbf{SecEx}, \emptyset\rangle, \epsilon\rangle & \text{if } s \xrightarrow{\langle \ominus,u,p,u'\rangle}_f s' \wedge secEx(s') \end{cases}$$

Figure 10: Definition of the $[\![q]\!](s,u)$ function – part 1.

$$[\![q]\!](s,u) = \begin{cases} \langle s', r, \epsilon\rangle & \text{if } s \xrightarrow{\langle u,\texttt{INSERT},T,\bar{t}\rangle}_f s' \wedge triggers(s') = \epsilon \\ \langle s', r, \epsilon\rangle & \begin{array}{l} \text{if } \exists s_1, \ldots, s_n \in \Omega_M.\ \exists t_1, \ldots, t_n \in \mathcal{TRIGGER}. \\ \quad s \xrightarrow{\langle u,\texttt{INSERT},T,\bar{t}\rangle}_f s_1 \xrightarrow{t_1}_f s_2 \xrightarrow{t_2}_f \ldots s_n \xrightarrow{t_n}_f s' \wedge \\ \quad r = res(s_1) \wedge Ex(s_1) = \emptyset \wedge \neg secEx(s_1) \wedge \\ \quad \bigwedge_{1 \leq i < n} (Ex(s_{i+1}) = \emptyset \wedge acA(s_{i+1}) \wedge acC(s_{i+1})) \wedge \\ \quad triggers(s') = \epsilon \wedge \neg secEx(s') \wedge Ex(s') = \emptyset \end{array} \\ \langle s', \dagger, \langle \mathbf{SecEx}, \emptyset\rangle\rangle & \text{if } s \xrightarrow{\langle u,\texttt{INSERT},T,\bar{t}\rangle}_f s' \wedge secEx(s') \wedge triggers(s') = \epsilon \\ \langle s', \dagger, \langle \mathbf{IntEx}, Ex(s')\rangle\rangle & \text{if } s \xrightarrow{\langle u,\texttt{INSERT},T,\bar{t}\rangle}_f s' \wedge Ex(s') \neq \emptyset \wedge triggers(s') = \epsilon \\ \langle s', \dagger, \langle t, \texttt{B}, \mathbf{IntEx}, Ex(s')\rangle\rangle & \begin{array}{l} \text{if } \exists s_1, \ldots, s_n \in \Omega_M.\ \exists t_1, \ldots, t_{n-1} \in \mathcal{TRIGGER}. \\ \quad s \xrightarrow{\langle u,\texttt{INSERT},T,\bar{t}\rangle}_f s_1 \xrightarrow{t_1}_f s_2 \xrightarrow{t_2}_f \ldots \xrightarrow{t_{n-1}}_f s_n \xrightarrow{t}_f s' \wedge \\ \quad Ex(s_1) = \emptyset \wedge \neg secEx(s_1) \wedge \\ \quad \bigwedge_{1 \leq i < n} (Ex(s_{i+1}) = \emptyset \wedge acA(s_{i+1}) \wedge acC(s_{i+1})) \wedge \\ \quad acA(s') \wedge acC(s') \wedge Ex(s') \neq \emptyset \wedge triggers(s') = \epsilon \end{array} \\ \langle s', \dagger, \langle t, \texttt{W}, \mathbf{SecEx}, \emptyset\rangle\rangle & \begin{array}{l} \text{if } \exists s_1, \ldots, s_n \in \Omega_M.\ \exists t_1, \ldots, t_{n-1} \in \mathcal{TRIGGER}. \\ \quad s \xrightarrow{\langle u,\texttt{INSERT},T,\bar{t}\rangle}_f s_1 \xrightarrow{t_1}_f s_2 \xrightarrow{t_2}_f \ldots \xrightarrow{t_{n-1}}_f s_n \xrightarrow{t}_f s' \wedge \\ \quad Ex(s_1) = \emptyset \wedge \neg secEx(s_1) \wedge \\ \quad \bigwedge_{1 \leq i < n} (Ex(s_{i+1}) = \emptyset \wedge acA(s_{i+1}) \wedge acC(s_{i+1})) \wedge \\ \quad \neg acC(s') \wedge triggers(s') = \epsilon \end{array} \\ \langle s', \dagger, \langle t, \texttt{B}, \mathbf{SecEx}, \emptyset\rangle\rangle & \begin{array}{l} \text{if } \exists s_1, \ldots, s_n \in \Omega_M.\ \exists t_1, \ldots, t_{n-1} \in \mathcal{TRIGGER}. \\ \quad s \xrightarrow{\langle u,\texttt{INSERT},T,\bar{t}\rangle}_f s_1 \xrightarrow{t_1}_f s_2 \xrightarrow{t_2}_f \ldots \xrightarrow{t_{n-1}}_f s_n \xrightarrow{t}_f s' \wedge \\ \quad Ex(s_1) = \emptyset \wedge \neg secEx(s_1) \wedge \\ \quad \bigwedge_{1 \leq i < n} (Ex(s_{i+1}) = \emptyset \wedge acA(s_{i+1}) \wedge acC(s_{i+1})) \wedge \\ \quad acC(s') \wedge \neg acA(s') \wedge triggers(s') = \epsilon \end{array} \end{cases}$$

Figure 11: Definition of the $[\![q]\!](s,u)$ function – part 2. Note that $q = \texttt{INSERT}\ \bar{t}\ \texttt{INTO}\ T$.

For INSERT and DELETE queries, $[\![q]\!](s,u)$ accounts for the execution of triggers as well. It also relies on the functions acC and acA that take as input a runtime state and retrieve the access control decisions associated with the trigger's condition and the trigger's action. We refer the reader to [24] for a formalization of these functions.

$$
[\![q]\!](s,u) = \begin{cases}
\langle s', r, \epsilon\rangle & \text{if } s \xrightarrow{\langle u, \mathtt{DELETE}, T, \bar{t}\rangle}_f s' \wedge \mathit{secEx}(s') \\
\langle s', r, \epsilon\rangle & \text{if } \exists s_1, \ldots, s_n \in \Omega_M.\ \exists t_1, \ldots, t_n \in \mathcal{TRIGGER}. \\
 & \quad s \xrightarrow{\langle u, \mathtt{DELETE}, T, \bar{t}\rangle}_f s_1 \xrightarrow{t_1}_f s_2 \xrightarrow{t_2}_f \ldots s_n \xrightarrow{t_n}_f s' \wedge \\
 & \quad r = \mathit{res}(s_1) \wedge \mathit{Ex}(s_1) = \emptyset \wedge \neg \mathit{secEx}(s_1) \wedge \\
 & \quad \bigwedge_{1 \leq i < n} (\mathit{Ex}(s_{i+1}) = \emptyset \wedge \mathit{acA}(s_{i+1}) \wedge \mathit{acC}(s_{i+1})) \wedge \\
 & \quad \mathit{triggers}(s') = \epsilon \wedge \neg \mathit{secEx}(s') \wedge \mathit{Ex}(s') = \emptyset \\
\langle s', \dagger, \langle \mathbf{SecEx}, \emptyset\rangle\rangle & \text{if } s \xrightarrow{\langle u, \mathtt{DELETE}, T, \bar{t}\rangle}_f s' \wedge \mathit{secEx}(s') \wedge \mathit{triggers}(s') = \epsilon \\
\langle s', \dagger, \langle \mathbf{IntEx}, \mathit{Ex}(s')\rangle\rangle & \text{if } s \xrightarrow{\langle u, \mathtt{DELETE}, T, \bar{t}\rangle}_f s' \wedge \mathit{Ex}(s') \neq \emptyset \wedge \mathit{triggers}(s') = \epsilon \\
\langle s', \dagger, \langle t, \mathtt{B}, \mathbf{IntEx}, \mathit{Ex}(s')\rangle\rangle & \text{if } \exists s_1, \ldots, s_n \in \Omega_M.\ \exists t_1, \ldots, t_{n-1} \in \mathcal{TRIGGER}. \\
 & \quad s \xrightarrow{\langle u, \mathtt{DELETE}, T, \bar{t}\rangle}_f s_1 \xrightarrow{t_1}_f s_2 \xrightarrow{t_2}_f \ldots \xrightarrow{t_{n-1}}_f s_n \xrightarrow{t}_f s' \wedge \\
 & \quad \mathit{Ex}(s_1) = \emptyset \wedge \neg \mathit{secEx}(s_1) \wedge \\
 & \quad \bigwedge_{1 \leq i < n} (\mathit{Ex}(s_{i+1}) = \emptyset \wedge \mathit{acA}(s_{i+1}) \wedge \mathit{acC}(s_{i+1})) \wedge \\
 & \quad \mathit{acA}(s') \wedge \mathit{acC}(s') \wedge \mathit{Ex}(s') \neq \emptyset \wedge \mathit{triggers}(s') = \epsilon \\
\langle s', \dagger, \langle t, \mathtt{W}, \mathbf{SecEx}, \emptyset\rangle\rangle & \text{if } \exists s_1, \ldots, s_n \in \Omega_M.\ \exists t_1, \ldots, t_{n-1} \in \mathcal{TRIGGER}. \\
 & \quad s \xrightarrow{\langle u, \mathtt{DELETE}, T, \bar{t}\rangle}_f s_1 \xrightarrow{t_1}_f s_2 \xrightarrow{t_2}_f \ldots \xrightarrow{t_{n-1}}_f s_n \xrightarrow{t}_f s' \wedge \\
 & \quad \mathit{Ex}(s_1) = \emptyset \wedge \neg \mathit{secEx}(s_1) \wedge \\
 & \quad \bigwedge_{1 \leq i < n} (\mathit{Ex}(s_{i+1}) = \emptyset \wedge \mathit{acA}(s_{i+1}) \wedge \mathit{acC}(s_{i+1})) \wedge \\
 & \quad \neg \mathit{acC}(s') \wedge \mathit{triggers}(s') = \epsilon \\
\langle s', \dagger, \langle t, \mathtt{B}, \mathbf{SecEx}, \emptyset\rangle\rangle & \text{if } \exists s_1, \ldots, s_n \in \Omega_M.\ \exists t_1, \ldots, t_{n-1} \in \mathcal{TRIGGER}. \\
 & \quad s \xrightarrow{\langle u, \mathtt{DELETE}, T, \bar{t}\rangle}_f s_1 \xrightarrow{t_1}_f s_2 \xrightarrow{t_2}_f \ldots \xrightarrow{t_{n-1}}_f s_n \xrightarrow{t}_f s' \wedge \\
 & \quad \mathit{Ex}(s_1) = \emptyset \wedge \neg \mathit{secEx}(s_1) \wedge \\
 & \quad \bigwedge_{1 \leq i < n} (\mathit{Ex}(s_{i+1}) = \emptyset \wedge \mathit{acA}(s_{i+1}) \wedge \mathit{acC}(s_{i+1})) \wedge \\
 & \quad \mathit{acC}(s') \wedge \neg \mathit{acA}(s') \wedge \mathit{triggers}(s') = \epsilon
\end{cases}
$$

Figure 12: Definition of the $[\![q]\!](s,u)$ function – part 3. Note that $q = \mathtt{DELETE}\ \bar{t}\ \mathtt{FROM}\ T$.

$$
[\![T \oplus \overline{t}]\!](s,u) = \begin{cases} \langle s'', \mathit{res}(s'), \epsilon \rangle & \text{if } s \xrightarrow{\langle u, \texttt{INSERT}, T, \overline{t} \rangle}_f s' \wedge \neg \mathit{secEx}(s') \wedge s = \langle db, U, S, T, V, \mathit{ctx} \rangle \wedge \\ & s'' = \langle db[R \oplus \overline{t}], U, S, T, V, \mathit{ctx} \rangle \\ \langle s, \dagger, \langle \mathbf{SecEx}, \emptyset \rangle \rangle & \text{if } s \xrightarrow{\langle u, \texttt{INSERT}, T, \overline{t} \rangle}_f s' \wedge \mathit{secEx}(s') \end{cases}
$$

$$
[\![T \ominus \overline{t}]\!](s,u) = \begin{cases} \langle s'', \mathit{res}(rs'), \epsilon \rangle & \text{if } s \xrightarrow{\langle u, \texttt{DELETE}, T, \overline{t} \rangle}_f s' \wedge \neg \mathit{secEx}(s') \wedge s = \langle db, U, S, T, V, \mathit{ctx} \rangle \wedge \\ & s'' = \langle db[R \ominus \overline{t}], U, S, T, V, \mathit{ctx} \rangle \\ \langle s, \dagger, \langle \mathbf{SecEx}, \emptyset \rangle \rangle & \text{if } s \xrightarrow{\langle u, \texttt{DELETE}, T, \overline{t} \rangle}_f s' \wedge \mathit{secEx}(s') \end{cases}
$$

Figure 13: Definition of the $[\![q]\!](s,u)$ function – part 4.

Appendix F. Enforcement – Extended version

F.1. Enforcement Operational Semantics

Here we provide the full operational semantics of our security monitor. Again, in the following atk denotes the user identifier associated with the attacker and sec_0 denotes the initial security policy.

Preliminaries. The auxiliary function $\mathit{atomic}(c)$ takes as input a WHILESQL program and returns $\top$ if there are c', c'' such that $c = [c']$ or $c = [c']\ ;\ c''$. The auxiliary function $\mathit{query}(c)$ takes as input an extended WHILESQL program and returns $\top$ if there are x, q such that $c = x \leftarrow q$ or $c = \|x \leftarrow q\|$. Finally, Figure 13 illustrates how the queries of the form $T \otimes \overline{v}$, where $\otimes \in \{\oplus, \ominus\}$, are handled by the underlying database.

Relaxed no-sensitive upgrade checks. Our enforcement mechanism is a dynamic security monitor. A common technique for preventing implicit leaks of sensitive information in this setting is using *no-sensitive upgrade* (NSU) checks [49]. Intuitively, a NSU check restricts the changes only to the variables whose label is at least that of the current execution's context, i.e., only for variables x such that $\Delta(\mathtt{pc}_u) \sqsubseteq \Delta(x)$. This guarantees that changes to "low" variables never happen in "high" execution contexts. NSU checks, however, are rather restrictive: they may block executions that do not leak information. This is particularly relevant for security lattices with many labels, such as our disclosure lattice, where many labels are simply unrelated and NSU checks often fail.

To address this, we propose a simple relaxation of NSU checks that still prevents leaks of sensitive information. In particular, our relaxed NSU checks exploit the fact that the initial security policy sec_0 can be used to determine which labels can be considered as permanently "low", no matter how the policy is modified during the execution. In more detail, given a variable x, the relaxed NSU check is defined as follows: $\Delta(\mathtt{pc}_u) \sqsubseteq \mathit{cl}(\mathit{auth}(\mathit{sec}_0, \mathit{atk})) \vee \Delta(\mathtt{pc}_u) \sqsubseteq \Delta(x)$, where sec_0 is the initial security policy. Our relaxed NSU check is satisfied whenever the standard NSU check is. Additionally, our relaxed NSU check allows flows of information whenever the security context $\mathtt{pc}_u$ is permanently low, i.e., $\Delta(\mathtt{pc}_u) \sqsubseteq \mathit{cl}(\mathit{auth}(\mathit{sec}_0, \mathit{atk}))$. In the following, we denote by $\mathbf{nsu}(x, \mathtt{pc})$ the predicate $\Delta(\mathtt{pc}) \sqsubseteq \mathit{cl}(\mathit{auth}(\mathit{sec}_0, \mathit{atk})) \vee \Delta(\mathtt{pc}) \sqsubseteq \Delta(x)$, where x is either a variable identifier or a predicate query, $\mathtt{pc}$ is an identifier of the form $\mathtt{pc}_u$, and u is a user identifier.

To illustrate, our relaxed NSU check allows changes to a variable x whose label is, say, $\mathit{cl}(T(1))$ in an execution context such that $\Delta(\mathtt{pc}_u) = \mathit{cl}(V(2))$ whenever the attacker atk is authorized to read both the table T and V with respect to the initial policy sec_0. A standard NSU check, instead, would have prevented the assignment since $\mathit{cl}(V(1)) \not\sqsubseteq \mathit{cl}(T(1))$.

Enforcement Rules. The relation $\rightsquigarrow_u$, where u is a user in UID, shown in Figures 14–17, formalizes the local operational semantics of our dynamic monitor. Figure 18 presents the security monitor rules for the global semantics. These rules rely on the auxiliary function $\mathit{updateLabel}(c)$, which takes as input a WHILESQL program c and returns $\top$ iff the first statement in c is a statement of the form **set pc to** l. The rules F-EVAL-STEP and F-EVAL-END are similar to the WHILESQL semantics. Additionally, the monitor uses the F-ATOMIC-STATEMENT rule to handle the atomic execution of code. Observe that the atomic execution does not consume the scheduler.

F.2. Expansion Process

Here we illustrate our expansion process for queries.

Extracting triggers. Our expansion process uses the function $\mathit{triggers} : \Omega_M \times \mathcal{Q} \times \mathcal{U} \rightarrow (\mathcal{TRIGGER} \times \mathcal{U} \times \mathcal{Q} \times \mathcal{Q} \times \mathcal{U} \times \mathcal{U})^*$ that provides an interface to the database and returns the triggers in the form of tuples $\langle t, u, \mathit{cond}, \mathit{act}, \mathit{invk}, \mathit{owner} \rangle$, where t is the trigger's identifier, u specifies the user under which privileges the triggers is to be executed (i.e., u is either the trigger's owner or the trigger's invoker depending on the trigger's definition), cond specifies t's `WHEN` condition, act is t's action, invk is the user that fired the trigger, and owner is the trigger's owner. Note that the variables associated with the tuple in the original command have already been replaced in both cond and act. Therefore, if the original command contains program variables, then cond and act may both contain program variables. Observe that the $\mathit{triggers}$ function can be implemented on top of the functions provided in [24]. Given a trigger represented as $t = \langle t, u, \mathit{cond}, \mathit{act}, \mathit{invk}, \mathit{owner} \rangle$, we denote by $\mathit{id}(t)$ the identifier id, by $\mathit{user}(t)$ the user u, by $\mathit{cond}(t)$ the condition cond, by $\mathit{act}(t)$ the action act, by $\mathit{invoker}(t)$ the invoker invk, and by $\mathit{owner}(t)$ the user owner.

F-SKIP
$$\frac{}{\langle \Delta, \mathbf{skip}, m, s\rangle \rightsquigarrow_u \langle \Delta, \varepsilon, m, s\rangle}$$

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

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-SEQ
$$\frac{\langle \Delta, c_1, m, s\rangle \overset{\mathcal{I}}{\rightsquigarrow}_u \langle \Delta', c_1', m', s'\rangle}{\langle \Delta, c_1 \,;\, c_2, m, s\rangle \overset{\mathcal{I}}{\rightsquigarrow}_u \langle \Delta', c_1' \,;\, c_2, m', s'\rangle}$$

F-SEQEMPTY
$$\frac{}{\langle \Delta, \varepsilon \,;\, c, m, s\rangle \rightsquigarrow_u \langle \Delta, c, m, s\rangle}$$

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

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

F-WHILETRUE
$$\frac{\begin{array}{c}[\![e]\!](m) = \mathbf{tt}\\ c' = [c \,;\, \mathbf{while}\ e\ \mathbf{do}\ c \,;\, \mathbf{set\ pc\ to}\ \Delta(\mathtt{pc}_u)]\\ \Delta' = \Delta[\mathtt{pc}_u \mapsto \Delta(e) \sqcup \Delta(\mathtt{pc}_u)]\end{array}}{\langle \Delta, \mathbf{while}\ e\ \mathbf{do}\ c, m, s\rangle \rightsquigarrow_u \langle \Delta', c', m, s\rangle}$$

F-WHILEFALSE
$$\frac{\begin{array}{c}[\![e]\!](m) = \mathbf{ff}\\ c' = [\mathbf{set\ pc\ to}\ \Delta(\mathtt{pc}_u)]\\ \Delta' = \Delta[\mathtt{pc}_u \mapsto \Delta(e) \sqcup \Delta(\mathtt{pc}_u)]\end{array}}{\langle \Delta, \mathbf{while}\ e\ \mathbf{do}\ c, m, s\rangle \rightsquigarrow_u \langle \Delta', c', m, s\rangle}$$

Figure 14: Security monitor – local operational semantics for assignments, out, and control flow statements.

F-EXPANDEDCODE
$$\frac{\langle \Delta, c, m, s\rangle \overset{\mathcal{I}}{\rightsquigarrow}_u \langle \Delta', c', m', s'\rangle}{\langle \Delta, [c], m, s\rangle \overset{\mathcal{I}}{\rightsquigarrow}_u \langle \Delta', [c'], m', s'\rangle}$$

F-REMOVEEXPANDEDCODE
$$\frac{}{\langle \Delta, [\varepsilon], m, s\rangle \rightsquigarrow_u \langle \Delta, \varepsilon, m, s\rangle}$$

Figure 15: Security monitor – local operational semantics for atomic statements.

Instrumented Commands. An instrumented command is a pair $\langle c, r\rangle$ such that c is an `INSERT` command, a `DELETE` command, or a trigger (represented as specified above using a 4-tuple), and $r \in \{\mathtt{ok}, \mathtt{dis}, \mathtt{secEx}, \mathtt{ex}\}$. Note that both commands and triggers may contain program variables. Given an instrumented command $\langle c, r\rangle$, $\mathit{first}(\langle c, r\rangle) = c$ and $\mathit{second}(\langle c, r\rangle) = r$.

Constructing the execution paths. Let $\bar{t}$ be a sequence of commands and triggers. We denote by $\mathit{paths}(\bar{t})$ the following function:

$$\begin{aligned}
&\mathit{paths}(\epsilon) = \emptyset\\
&\mathit{paths}(c) = \{\langle c, \mathtt{ok}\rangle, \langle c, \mathtt{secEx}\rangle, \langle c, \mathtt{ex}\rangle\} \text{ where } c \text{ is an SQL command}\\
&\mathit{paths}(t) = \{\langle t, \mathtt{ok}\rangle, \langle t, \mathtt{dis}\rangle, \langle t, \mathtt{secEx}\rangle, \langle t, \mathtt{ex}\rangle\} \text{ where } t \text{ is a trigger}\\
&\mathit{paths}(t \cdot \bar{t}) = \mathit{merge}(\mathit{paths}(t), \mathit{paths}(\bar{t}))\\
&\mathit{merge}(S_1, S_2) = \{s_1 \cdot s_2 \mid s_1 \in S_1 \wedge s_2 \in S_2 \wedge \neg\exists s_1', o.\ (s_1 = s_1' \cdot \langle o, \mathtt{ex}\rangle \vee s_1 = s_1' \cdot \langle o, \mathtt{secEx}\rangle)\}
\end{aligned}$$

Configuration-consistent Execution Paths. The execution paths computed through the paths function may, in general, contain unfeasible paths. For instance, they may contain commands terminating in a security exception even though this may not happen given the current security policy. To take this into account, we define the notion of a configuration-consistent execution path. Note that there is no analogous of configuration-consistent path for integrity exceptions caused

F-UPDATELABELS
$$\frac{}{\langle \Delta, \mathbf{set\ pc\ to}\ l, m, s\rangle \rightsquigarrow_u \langle \Delta[\mathtt{pc}_u \mapsto l], \varepsilon, m, s\rangle}$$

F-ASUSER
$$\frac{\mathit{query}(c) \qquad \langle \Delta[\mathtt{pc}_{u'} \mapsto \Delta(\mathtt{pc}_u)], c, m, s\rangle \rightsquigarrow_{u'} \langle \Delta', c', m', s'\rangle}{\langle \Delta, \mathbf{asuser}(u', c), m, s\rangle \rightsquigarrow_u \langle \Delta'[\mathtt{pc}_{u'} \mapsto \Delta(\mathtt{pc}_{u'})], c', m', s'\rangle}$$

Figure 16: Security monitor – local operational semantics for **set pc** and **asuser** statements.

F-EXPAND

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

F-SELECT

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

F-UPDATEDATABASEOK

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

F-UPDATECONFIGURATIONOK

$$\frac{\begin{array}{c}\{v_1, \ldots, v_n\} = \mathit{vars}(q) \qquad q' = q[v_1 \mapsto [\![v_1]\!](m), \ldots, v_n \mapsto [\![v_n]\!](m)] \qquad \mathit{isCfgCmd}(q') \qquad [\![q']\!](s, u) = \langle s', r, \epsilon\rangle \\ \ell_{\mathit{cmd}} = \bigsqcup_{1 \leq i \leq n} \Delta(v_i) \qquad \ell_{\mathit{cmd}} \sqsubseteq \mathit{cl}(\mathit{auth}(\mathit{sec}_0, \mathit{atk})) \qquad \Delta(\mathtt{pc}_u) \sqsubseteq \mathit{cl}(\mathit{auth}(\mathit{sec}_0, \mathit{atk})) \qquad \mathbf{nsu}(x, \mathtt{pc}_u)\end{array}}{\langle \Delta, \|x \leftarrow q\|, m, s\rangle \overset{\langle \mathit{public}, q'\rangle}{\rightsquigarrow}_u \langle \Delta[x \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_{\mathit{cmd}}], \varepsilon, m[x \mapsto r], s'\rangle}$$

Figure 17: Security monitor – local operational semantics for database operations.

F-EVAL-STEP

$$\frac{\begin{array}{c}\forall i \in \{1, \ldots, |C|\}, u' \in \mathit{UID}.\ C|_n \neq \langle u', \varepsilon\rangle \qquad |C| = |M| \\ n = 1 + (n'\ \mathbf{mod}\ |C|) \qquad C|_n = \langle u, c\rangle \qquad M|_n = \langle u, m\rangle \qquad \langle \Delta, c, m, s\rangle \overset{\tau}{\rightsquigarrow}_u \langle \Delta', c', m', s'\rangle \quad \neg\mathit{atomic}(c') \\ C' = C|_1 \cdot \ldots \cdot C|_{n-1} \cdot \langle u, c'\rangle \cdot C|_{n+1} \cdot \ldots \cdot C|_{|C|} \qquad M' = M|_1 \cdot \ldots \cdot M|_{n-1} \cdot \langle u, m'\rangle \cdot M|_{n+1} \cdot \ldots \cdot M|_{|C|}\end{array}}{\langle \Delta, C, M, s, n' \cdot \mathcal{S}\rangle \overset{\tau}{\rightsquigarrow} \langle \Delta', C', M', s', \mathcal{S}\rangle}$$

F-ATOMIC-STATEMENT

$$\frac{\begin{array}{c}\forall i \in \{1, \ldots, |C|\}, u \in \mathit{UID}.\ C|_i \neq \langle u', \varepsilon\rangle \\ |C| = |M| \qquad n = 1 + (n'\ \mathbf{mod}\ |C|) \qquad C|_n = \langle u, c\rangle \qquad M|_n = \langle u, m\rangle \qquad \langle \Delta, c, m, s\rangle \overset{\tau}{\rightsquigarrow}_u \langle \Delta', c', m', s'\rangle \\ \mathit{atomic}(c') \qquad C' = C|_1 \cdot \ldots \cdot C|_{n-1} \cdot \langle u, c'\rangle \cdot C|_{n+1} \cdot \ldots \cdot C|_{|C|} \qquad M' = M|_1 \cdot \ldots \cdot M|_{n-1} \cdot \langle u, m'\rangle \cdot M|_{n+1} \cdot \ldots \cdot M|_{|C|}\end{array}}{\langle \Delta, C, M, s, n' \cdot \mathcal{S}\rangle \overset{\tau}{\rightsquigarrow} \langle \Delta', C', M', s', n' \cdot \mathcal{S}\rangle}$$

F-EVAL-END

$$\frac{\begin{array}{c}1 \leq n \leq |C| \qquad \forall n' < n, u' \in \mathit{UID}.\ C|_{n'} \neq \langle u', \varepsilon\rangle \qquad C|_n = \langle u, \varepsilon\rangle \\ |C| = |M| \qquad C' = C|_1 \cdot \ldots \cdot C|_{n-1} \cdot C|_{n+1} \cdot \ldots \cdot C|_{|C|} \qquad M' = M|_1 \cdot \ldots \cdot M|_{n-1} \cdot M|_{n+1} \cdot \ldots \cdot M|_{|C|}\end{array}}{\langle \Delta, C, M, s, \mathcal{S}\rangle \xrightarrow{\tau} \langle \Delta, C', M', s, \mathcal{S}\rangle}$$

Figure 18: Security monitor – global operational semantics.

by `INSERT` and `DELETE` commands. The reason is that these exceptions depend on the database content and are directly handled by the expansion process.

In the following, let s be a runtime state, m be a memory, u be a user, $\overline{t}$ be a sequence of instrumented commands (i.e., an execution path), c be a database command, and t be a trigger. Furthermore, given a command (or trigger) o containing program variables, we denote by $o(m)$ the command (or trigger) obtained from o by replacing all program variables with the corresponding values in m. The configuration-consistency relation is defined as follows:

1) $s, m, u \models^{\mathit{cfg}} \langle c, \mathtt{ok}\rangle$ if $\mathit{allowed}(s, u, c(m))$.
2) $s, m, u \models^{\mathit{cfg}} \langle c, \mathtt{ex}\rangle$ if $\mathit{allowed}(s, u, c(m))$ and c is an `INSERT` or `DELETE` command.
3) $s, m, u \models^{\mathit{cfg}} \langle c, \mathtt{secEx}\rangle$ if $\neg\mathit{allowed}(s, u, c(m))$.
4) $s, m, u \models^{\mathit{cfg}} \langle t, \mathtt{ok}\rangle$ if $\mathit{allowed}(s, u, t(m))$.
5) $s, m, u \models^{\mathit{cfg}} \langle t, \mathtt{ex}\rangle$ if $\mathit{allowed}(s, u, t(m))$ and t's action is an `INSERT` or `DELETE` command.
6) $s, m, u \models^{\mathit{cfg}} \langle t, \mathtt{secEx}\rangle$ if $\neg\mathit{allowed}(s, u, t(m))$.
7) $s, m, u \models^{\mathit{cfg}} \langle t, \mathtt{dis}\rangle$.
8) $s, m, u \models^{\mathit{cfg}} \langle o, r\rangle \cdot \overline{t}$ iff $\overline{t} \neq \epsilon$, $s, m, u \models^{\mathit{cfg}} \langle o, r\rangle$, $r \neq \mathtt{dis}$, and $\mathit{apply}(s, o(m)), m, u \models^{\mathit{cfg}} \overline{t}$.
9) $s, m, u \models^{\mathit{cfg}} \langle o, \mathtt{dis}\rangle \cdot \overline{t}$ iff $\overline{t} \neq \epsilon$, $s, m, u \models^{\mathit{cfg}} \langle o, \mathtt{dis}\rangle$ and $s, m, u \models^{\mathit{cfg}} \overline{t}$.

$$allowed(s,u,\texttt{SELECT }\phi) = \begin{cases} \top & \text{if } s \xrightarrow{\langle u,\texttt{SELECT},\phi\rangle}_f s' \wedge \neg secEx(s') \\ \bot & \text{if } s \xrightarrow{\langle u,\texttt{SELECT},\phi\rangle}_f s' \wedge secEx(s') \end{cases}$$

$$allowed(s,u,\texttt{GRANT } p \texttt{ TO } u') = \begin{cases} \top & \text{if } s \xrightarrow{\langle \oplus,u,p,u'\rangle}_f s' \wedge \neg secEx(s') \\ \bot & \text{if } s \xrightarrow{\langle \oplus,u,p,u'\rangle}_f s' \wedge secEx(s') \end{cases}$$

$$allowed(s,u,\texttt{GRANT } p \texttt{ TO } u' \texttt{ WITH GRANT OPTION}) = \begin{cases} \top & \text{if } s \xrightarrow{\langle \oplus^*,u,p,u'\rangle}_f s' \wedge \neg secEx(s') \\ \bot & \text{if } s \xrightarrow{\langle \oplus^*,u,p,u'\rangle}_f s' \wedge secEx(s') \end{cases}$$

$$allowed(s,u,\texttt{REVOKE } p \texttt{ FROM } u') = \begin{cases} \top & \text{if } s \xrightarrow{\langle \ominus,u,p,u'\rangle}_f s' \wedge \neg secEx(s') \\ \bot & \text{if } s \xrightarrow{\langle \ominus,u,p,u'\rangle}_f s' \wedge secEx(s') \end{cases}$$

$$allowed(s,u,\texttt{INSERT } \overline{v} \texttt{ INTO } T) = \begin{cases} \top & \text{if } s \xrightarrow{\langle u,\texttt{INSERT},T,\overline{v}\rangle}_f s' \wedge \neg secEx(s') \\ \bot & \text{if } s \xrightarrow{\langle u,\texttt{INSERT},T,\overline{v}\rangle}_f s' \wedge secEx(s') \end{cases}$$

$$allowed(s,u,\texttt{DELETE } \overline{v} \texttt{ FROM } T) = \begin{cases} \top & \text{if } s \xrightarrow{\langle u,\texttt{DELETE},T,\overline{v}\rangle}_f s' \wedge \neg secEx(s') \\ \bot & \text{if } s \xrightarrow{\langle u,\texttt{DELETE},T,\overline{v}\rangle}_f s' \wedge secEx(s') \end{cases}$$

$$allowed(s,u,\texttt{CREATE } obj) = \begin{cases} \top & \text{if } s \xrightarrow{\langle u,\texttt{CREATE},obj\rangle}_f s' \wedge \neg secEx(s') \\ \bot & \text{if } s \xrightarrow{\langle u,\texttt{CREATE},obj\rangle}_f s' \wedge secEx(s') \end{cases}$$

$$allowed(s,u,t) = allowed(s, owner(t), act(t)) \text{ if } t \text{ is a trigger with owner's privileges}$$

Figure 19: *allowed* function for the expansion process.

$apply(s,u,\texttt{SELECT }\phi) = s$
$apply(s,u,\texttt{GRANT } p \texttt{ TO } u') = s'$ where $s \xrightarrow{\langle \oplus,u',p,u\rangle}_f s'$
$apply(s,u,\texttt{GRANT } p \texttt{ TO } u' \texttt{ WITH GRANT OPTION}) = s'$ where $s \xrightarrow{\langle \oplus^*,u',p,u\rangle}_f s'$
$apply(s,u,\texttt{REVOKE } p \texttt{ FROM } u') = s'$ where $s \xrightarrow{\langle \ominus,u',p,u\rangle}_f s'$
$apply(s,u,\texttt{INSERT } \overline{v} \texttt{ INTO } T) = s$
$apply(s,u,\texttt{DELETE } \overline{v} \texttt{ FROM } T) = s$
$apply(s,u,\texttt{CREATE } obj) = s'$ where $s \xrightarrow{\langle u,\texttt{CREATE},obj\rangle}_f s'$
$apply(s,u,t) = apply(s, user(t), act(t))$ if t is a trigger

Figure 20: *apply* function for the expansion process. Note that we are interested only in changes to the database configuration, not to the database state. Therefore, the function does not update the database on INSERT and DELETE commands.

The above definition relies on the functions *allowed* and *apply*, which are formalized in Figures 19 and 20. Let $\overline{t}$ be a sequence of commands and triggers (possibly containing program variables), s be a database state, m be a memory, and u be a user. We denote by $consPaths(\overline{t}, s, ,u)$ the set of all configuration-consistent paths derivable from $\overline{t}$. Formally, $consPaths(\overline{t}, s, m, u) = \{\overline{t'} \in paths(\overline{t}) \mid s, m, u \models^{cfg} \overline{t'}\}$.

Weakest precondition for database updates. We now introduce weakest preconditions for INSERT and DELETE operations on databases. In the following, we assume that constants are not used inside predicate symbols. E.g., the formula $T(\overline{v})$ is expressed as $\exists \overline{x}.\ T(\overline{x}) \wedge \overline{x} = \overline{v}$. Let ϕ be a relational calculus sentence that does not refer to views (for formulae that refer to views, one can first replace the views with their definitions and later compute the weakest precondition). Furthermore, we denote by $T \oplus \overline{v}$ (respectively $T \ominus \overline{v}$) an insertion (respectively deletion) operation on the database. Note that $\overline{v}$ may contain program variables. The weakest precondition of ϕ given $T \oplus \overline{v}$, written $wp(\phi, T \oplus \overline{v})$, is obtained by replacing all occurrences of $T(\overline{x})$ with $(T(\overline{x}) \vee \overline{x} = \overline{v})$. Similarly, the weakest precondition of ϕ given $T \ominus \overline{v}$, written $wp(\phi, T \ominus \overline{v})$, is obtained by replacing all occurrences of $T(\overline{x})$ with $(T(\overline{x}) \wedge \overline{x} \neq \overline{v})$.

Weakest precondition for execution paths. In Figure 21, we extend weakest preconditions from single INSERT and

$wp(\phi, \epsilon) = \phi$

$wp(\phi, \overline{t'} \cdot ic) = wp(wp(\phi, ic), \overline{t'})$

where ic is an instrumented command and $\overline{t'}$ is a sequence of instrumented commands

$wp(\phi, \langle \texttt{INSERT } T \texttt{ INTO } \overline{e}, \texttt{ok}\rangle) = wp(\phi, T \oplus \overline{e})$

$wp(\phi, \langle \texttt{INSERT } T \texttt{ INTO } \overline{e}, \texttt{ex}\rangle) = wp(\phi, T \oplus \overline{e})$

$wp(\phi, \langle \texttt{INSERT } T \texttt{ INTO } \overline{e}, \texttt{secEx}\rangle) = \phi$

$wp(\phi, \langle \texttt{DELETE } T \texttt{ FROM } \overline{e}, \texttt{ok}\rangle) = wp(\phi, T \ominus \overline{e})$

$wp(\phi, \langle \texttt{DELETE } T \texttt{ FROM } \overline{e}, \texttt{ex}\rangle) = wp(\phi, T \ominus \overline{e})$

$wp(\phi, \langle \texttt{DELETE } T \texttt{ FROM } \overline{e}, \texttt{secEx}\rangle) = \phi$

$wp(\phi, \langle c, r\rangle) = \phi$ where c is a `SELECT`, `GRANT`, `REVOKE`, or `CREATE` command

$wp(\phi, \langle t, \texttt{ok}\rangle) = wp(\phi, act(t))$ where t is a trigger

$wp(\phi, \langle t, \texttt{ex}\rangle) = wp(\phi, act(t))$ where t is a trigger

$wp(\phi, \langle t, \texttt{secEx}\rangle) = \phi$ where t is a trigger

$wp(\phi, \langle t, \texttt{dis}\rangle) = \phi$ where t is a trigger

Figure 21: Weakest precondition for sequences of instrumented commands.

Expansion Procedure.

$$expand(s, m, u, x \leftarrow q) = decls(s, m, u, x \leftarrow q) \; ; \; body(s, m, u, x \leftarrow q)$$

Shorthands.

$\overline{t}_q = triggers(s, u, q)$ $\quad$ $first(\langle a, b\rangle) = a$ $\quad$ $\overline{EP}_{s,m,u,q} = toList(consPaths(q \cdot \overline{t}_q, s, u))$

$\Gamma = \{\gamma_1, \ldots, \gamma_n\}$ are the integrity constraints $\quad$ $\overline{L_\Gamma} = toList(2^{\{\gamma_1, \ldots, \gamma_n\}})$

$throwsEx(\overline{t}) = \exists o. \; (\overline{t}|_{|\overline{t}|} = \langle o, \texttt{ex}\rangle \vee \overline{t}|_{|\overline{t}|} = \langle o, \texttt{secEx}\rangle)$ $\quad$ $secEx(\overline{t}) = \exists o. \; \overline{t}|_{|\overline{t}|} = \langle o, \texttt{secEx}\rangle$

$isCfgCmd(q) = \top$ iff q is an `GRANT`, `REVOKE`, `ADD USER`, or `CREATE` command

$isInsDel(q) = \top$ iff q is an `INSERT` or `DELETE` command

$isSelect(q) = \top$ iff q is a `SELECT` command

$isTrigger(o) = \top$ iff o is a trigger

Computing the expansion's auxiliary declarations.

$decls(s, m, u, x \leftarrow q) = \mathbf{;}(map(g_{s,m,u,q}, \overline{EP}_{s,m,u,q}))$

$g_{s,m,u,q}(\overline{t}) = h_{s,m,u,q,\overline{t}}(1); \ldots; h_{s,m,u,q,\overline{t}}(|\overline{t}|)$

$h_{s,m,u,q,\overline{t}}(i) = \|x^{\overline{t}}_{i,\gamma_1} \leftarrow \texttt{SELECT } wp(\gamma_1, \overline{t}^i)\|; \ldots; \|x^{\overline{t}}_{i,\gamma_n} \leftarrow \texttt{SELECT } wp(\gamma_n, \overline{t}^i)\|$

where $\overline{t}|_i$ is not a trigger and $x^{\overline{t}}_{i,\gamma_1}, \ldots, x^{\overline{t}}_{i,\gamma_n}$ are fresh variables

$h_{s,m,u,q,\overline{t}}(i) = \|x^{\overline{t}}_{i,cond} \leftarrow \texttt{SELECT } wp(\varphi, \overline{t}^{i-1})\|; \|x^{\overline{t}}_{i,\gamma_1} \leftarrow \texttt{SELECT } wp(\gamma_1, \overline{t}^i)\|; \ldots; \|x^{\overline{t}}_{i,\gamma_n} \leftarrow \texttt{SELECT } wp(\gamma_n, \overline{t}^i)\|$

where $\overline{t}|_i$ is a trigger, φ is $\overline{t}|_i$'s condition, and $x^{\overline{t}}_{i,cond}, x^{\overline{t}}_{i,\gamma_1}, \ldots, x^{\overline{t}}_{i,\gamma_n}$ are fresh variables

Figure 22: Expansion process – 1.

`DELETE` commands to sequences of instrumented commands.

Expansion procedure. Finally, the expansion procedure is shown in Figures 22 and 23. In the figures, s is a database state, m is a memory, u is a user identifier, and $x \leftarrow q$ is an SQL command. Without loss of generality, we assume that $x \notin free(q)$ (if this is not the case, one can just introduce an additional temporary variable). Furthermore, given a list $c_1 \cdot \ldots \cdot c_n$ of WHILESQL statements, we denote by $\mathbf{;}(c_1 \cdot \ldots \cdot c_n)$ the statement $c_1; c_2; \ldots; c_n$. Similarly, given a list of WHILESQL expressions $e_1 \cdot \ldots \cdot e_n$, we denote by $\bigwedge(e_1 \cdot \ldots \cdot e_n)$ the expression $e_1 \wedge e_2 \wedge \ldots \wedge e_n$.

Computing the expansion's body.

$$body(s, m, u, x \leftarrow q) = \textbf{;}(map(d_{s,m,u,x,q}, \overline{EP}_{s,m,u,x,q}))$$

$$d_{s,m,u,x,q}(\bar{t}) = \textbf{if } cond_{s,m,u,x,q}(\bar{t}) \textbf{ then } body_{s,m,u,x,q}(\bar{t}) \textbf{ else skip}$$

$$cond_{s,m,u,x,q}(\bar{t}) = \wedge(map(c_{s,m,u,x,q,\bar{t}}, 1 \cdot \ldots \cdot |\bar{t}|))$$

$$c_{s,m,u,x,q,\bar{t}}(i) = x^{\bar{t}}_{i,\gamma_1} \wedge \ldots \wedge x^{\bar{t}}_{i,\gamma_n} \text{ where } \bar{t}|_i = \langle c, \texttt{ok}\rangle \text{ and } c \text{ is not a trigger}$$

$$c_{s,m,u,x,q,\bar{t}}(i) = \neg(x^{\bar{t}}_{i,\gamma_1} \wedge \ldots \wedge x^{\bar{t}}_{i,\gamma_n}) \text{ where } \bar{t}|_i = \langle c, \texttt{ex}\rangle \text{ and } c \text{ is not a trigger}$$

$$c_{s,m,u,x,q,\bar{t}}(i) = \top \text{ where } \bar{t}|_i = \langle c, \texttt{secEx}\rangle \text{ and } c \text{ is not a trigger}$$

$$c_{s,m,u,x,q,\bar{t}}(i) = x^{\bar{t}}_{i,cond} \wedge x^{\bar{t}}_{i,\gamma_1} \wedge \ldots \wedge x^{\bar{t}}_{i,\gamma_n} \text{ where } \bar{t}|_i = \langle t, \texttt{ok}\rangle \text{ and } t \text{ is a trigger}$$

$$c_{s,m,u,x,q,\bar{t}}(i) = x^{\bar{t}}_{i,cond} \wedge \neg(x^{\bar{t}}_{i,\gamma_1} \wedge \ldots \wedge x^{\bar{t}}_{i,\gamma_n}) \text{ where } \bar{t}|_i = \langle t, \texttt{ex}\rangle \text{ and } t \text{ is a trigger}$$

$$c_{s,m,u,x,q,\bar{t}}(i) = x^{\bar{t}}_{i,cond} \text{ where } \bar{t}|_i = \langle t, \texttt{secEx}\rangle \text{ and } t \text{ is a trigger}$$

$$c_{s,m,u,x,q,\bar{t}}(i) = \neg x^{\bar{t}}_{i,cond} \text{ where } \bar{t}|_i = \langle t, \texttt{dis}\rangle \text{ and } t \text{ is a trigger}$$

$$body_{s,m,u,x,q}(\bar{t}) = \begin{cases} x = \langle \textbf{SecEx}, \emptyset\rangle & \text{if } \bar{t}|_1 = \langle o, \texttt{secEx}\rangle \text{ and} \\ & \quad o \text{ is not a trigger} \\ x = \langle id(\mathit{first}(\bar{t}|_{|\bar{t}|})), \textbf{SecEx}, \emptyset\rangle & \text{if } \bar{t}|_{|\bar{t}|} = \langle o, \texttt{secEx}\rangle \text{ and} \\ & \quad o \text{ is a trigger} \\ \textbf{;}(map(e_{s,m,u,x,q,\bar{t}}, \overline{L_\Gamma})) & \text{if } \bar{t}|_{|\bar{t}|} = \langle o, \texttt{ex}\rangle \\ \textbf{;}\,(map(b_{s,m,u,x,q,\bar{t}}, 1 \cdot \ldots \cdot |\bar{t}|)) & \text{otherwise} \end{cases}$$

$$b_{s,m,u,x,q,\bar{t}}(i) = \|x \leftarrow T \oplus \bar{e}\| \text{ where } \bar{t}|_i = \langle \texttt{INSERT } T \texttt{ INTO } \bar{e}, \texttt{ok}\rangle$$

$$b_{s,m,u,x,q,\bar{t}}(i) = \|x \leftarrow T \ominus \bar{e}\| \text{ where } \bar{t}|_i = \langle \texttt{DELETE } T \texttt{ FROM } \bar{e}, \texttt{ok}\rangle$$

$$b_{s,m,u,x,q,\bar{t}}(i) = \|x \leftarrow q'\| \text{ where } \bar{t}|_i = \langle q', \texttt{ok}\rangle, \mathit{isTrigger}(q') = \bot, \text{ and } \mathit{isInsDel}(q') = \bot$$

$$b_{s,m,u,x,q,\bar{t}}(i) = \textbf{asuser}(user(t), \|y \leftarrow T \oplus \bar{e}\|) \text{ where } \bar{t}|_i = \langle t, \texttt{ok}\rangle, t \text{ is a trigger,}$$
$$act(t) = \texttt{INSERT } T \texttt{ INTO } \bar{e}, \text{ and } y \text{ is a fresh variable}$$

$$b_{s,m,u,x,q,\bar{t}}(i) = \textbf{asuser}(user(t), \|y \leftarrow T \ominus \bar{e}\|) \text{ where } \bar{t}|_i = \langle t, \texttt{ok}\rangle, t \text{ is a trigger,}$$
$$act(t) = \texttt{DELETE } T \texttt{ FROM } \bar{e}, \text{ and } y \text{ is a fresh variable}$$

$$b_{s,m,u,x,q,\bar{t}}(i) = \textbf{asuser}(user(t), \|y \leftarrow act(t)\|) \text{ where } \bar{t}|_i = \langle t, \texttt{ok}\rangle, t \text{ is a trigger,}$$
$$\mathit{isInsDel}(act(t)) = \bot, \text{ and } y \text{ is a fresh variable}$$

$$b_{s,m,u,x,q,\bar{t}}(i) = \textbf{skip} \text{ where } \bar{t}|_i = \langle t, \texttt{dis}\rangle$$

$$e_{s,m,u,x,q,\bar{t}}(\Theta) = \textbf{if } \bigwedge_{\gamma \in \Theta} \neg x^{\bar{t}}_{|\bar{t}|,\gamma} \wedge \bigwedge_{\gamma \in \Gamma \setminus \Theta} x^{\bar{t}}_{|\bar{t}|,\gamma} \textbf{ then } e'_{s,m,u,x,q,\bar{t}}(\Theta) \textbf{ else skip}$$

$$e'_{s,m,u,x,q,\bar{t}}(\Theta) = \begin{cases} x = \langle \textbf{IntEx}, \Theta\rangle & \text{if } |\bar{t}| = 1 \\ x = \langle id(\mathit{first}(\bar{t}|_{|\bar{t}|})), \textbf{IntEx}, \Theta\rangle & \text{if } |\bar{t}| > 1 \end{cases}$$

Figure 23: Expansion process – 2.

Appendix G. Monitor's transparency

Here, we show that the monitor of §5 is transparent. In more detail, we prove that the monitor's local semantics is transparent. Furthermore, we also show that for sequential schedulers the monitor's global semantics is transparent as well.

G.1. Local semantics

Before proving the correctness of the local semantics, we introduce some terminology and notation. Given an extended WHILESQL program c, we denote by $\mathit{strip}(c)$ the program obtained by (1) removing statements of the form **set pc to** l, $\|x \leftarrow q\|$, and **asuser**(u, c') and (2) replacing $[c]$ with c. Furthermore, we write $\mathit{safe}(c)$ iff $\mathit{strip}(c)$ is not of the form $x \leftarrow q$, $x \leftarrow q \ ; c'$, **asuser**(u, c'), **asuser**$(u, c') \ ; c''$, $\|x \leftarrow q\|$, or $\|x \leftarrow q\| \ ; c''$.

In Lemma G.1 we prove the correctness of the weakest precondition operator for `INSERT` and `DELETE` commands.

Lemma G.1. *Let ϕ be a sentence that does not refer to views, m be a memory, and db a database state. For all well-formed assignments ν for ϕ, the following facts hold:*

1) *$[wp(\phi, T \oplus \overline{v}(m))\nu]^{db}$ holds iff $[\phi\nu]^{db'}$ holds, where $db'(R) = db(R)$ for all $R \neq T$ and $db'(T) = db(T) \cup \{\overline{v}(m)\}$.*
2) *$[wp(\phi, T \ominus \overline{v}(m))\nu]^{db}$ holds iff $[\phi\nu]^{db'}$ holds, where $db'(R) = db(R)$ for all $R \neq T$ and $db'(T) = db(T) \setminus \{\overline{v}(m)\}$.*
3) *$[\neg wp(\phi, c)\nu]^{db} = [wp(\neg\phi, c)\nu]^{db}$.*

Proof. Let ϕ be a sentence that does not refer to views, m be a memory, and db a database state.

Proof of (1). Let db' be the database state such that $db'(R) = db(R)$ for all $R \neq T$ and $db'(T) = db(T) \cup \{\overline{v}(m)\}$.

For the *if* direction, we assume that for all well-formed assignments ν for ϕ, $[wp(\phi, T \oplus \overline{v}(m))\nu]^{db} = \top$. We now prove, by structural induction on ϕ, that $[\phi\nu]^{db'} = \top$. There are two base cases:

- ϕ is $R(\overline{x})$. If $R \neq T$, then $wp(\phi, T \oplus \overline{v}(m)) = \phi$ and the claim trivially holds since $db(R) = db'(R)$. If $R = T$, then $wp(\phi, T \oplus \overline{v}(m)) = R(\overline{x}) \vee \overline{x} = \overline{v}(m)$. From this and $[wp(\phi, T \oplus \overline{v}(m))\nu]^{db} = \top$, it follows that $\nu(\overline{x}) \in db(T)$ or $\nu(\overline{x}) = \overline{v}(m)$. If $\nu(\overline{x}) \in db(T)$, then $\nu(\overline{x}) \in db'(T)$ as well. If $\nu(\overline{x}) = \overline{v}(m)$, then $\nu(\overline{x}) \in db'(T)$ by construction. Hence, $[\phi\nu]^{db'} = \top$.
- ϕ is $x_1 = x_2$. Then, $wp(\phi, T \oplus \overline{v}(m)) = \phi$ and the claim trivially holds.
- ϕ is $x_1 = c$. Then, $wp(\phi, T \oplus \overline{v}(m)) = \phi$ and the claim trivially holds.

For the induction step, we assume that the claim holds for all sub-formulae of ϕ and we show that it holds for ϕ as well. There are several cases:

- ϕ is $\psi \wedge \gamma$. From $[wp(\phi, T \oplus \overline{v}(m))\nu]^{db} = \top$ and $wp(\phi, T \oplus \overline{v}(m)) = wp(\psi, T \oplus \overline{v}(m)) \wedge wp(\gamma, T \oplus \overline{v}(m))$, it follows that $[wp(\psi, T \oplus \overline{v}(m))\nu]^{db} = \top$ and $[wp(\gamma, T \oplus \overline{v}(m))\nu]^{db} = \top$. From this, ν is a well-formed assignment for ψ and γ, and the induction hypothesis, it follows that $[\psi\nu]^{db'} = \top$ and $[\gamma\nu]^{db'} = \top$. From this, $[(\psi \wedge \gamma)\nu]^{db'} = \top$ and therefore $[\phi\nu]^{db'} = \top$.
- ϕ is $\psi \vee \gamma$. From $[wp(\phi, T \oplus \overline{v}(m))\nu]^{db} = \top$ and $wp(\phi, T \oplus \overline{v}(m)) = wp(\psi, T \oplus \overline{v}(m)) \vee wp(\gamma, T \oplus \overline{v}(m))$, it follows that $[wp(\psi, T \oplus \overline{v}(m))\nu]^{db} = \top$ or $[wp(\gamma, T \oplus \overline{v}(m))\nu]^{db} = \top$. From this, ν is a well-formed assignment for ψ and γ, and the induction hypothesis, it follows that $[\psi\nu]^{db'} = \top$ or $[\gamma\nu]^{db'} = \top$. From this, $[(\psi \vee \gamma)\nu]^{db'} = \top$ and therefore $[\phi\nu]^{db'} = \top$.
- ϕ is $\neg\psi$. From $[wp(\phi, T \oplus \overline{v}(m))\nu]^{db} = \top$ and $wp(\phi, T \oplus \overline{v}(m)) = \neg wp(\psi, T \oplus \overline{v}(m))$, it follows that $[wp(\psi, T \oplus \overline{v}(m))\nu]^{db} = \bot$. From this and the induction hypothesis, it follows that $[\psi\nu]^{db'} = \bot$. From this, $[(\neg\psi)\nu]^{db'} = \top$ and therefore $[\phi\nu]^{db'} = \top$.
- ϕ is $\exists x.\ \psi$. From $[wp(\phi, T \oplus \overline{v}(m))\nu]^{db} = \top$ and $wp(\phi, T \oplus \overline{v}(m)) = \exists x.\ wp(\psi, T \oplus \overline{v}(m))$, it follows that there is a value $v \in \mathbf{dom}$ such that $[wp(\psi, T \oplus \overline{v}(m))\nu[x \mapsto v]]^{db} = \top$. From this, $\nu[x \mapsto v]$ is a well-formed assignment for ψ, and the induction hypothesis, it follows that $[\psi\nu]^{db'} = \top$. From this, $[(\exists x.\ \psi)\nu]^{db'} = \top$ and therefore $[\phi\nu]^{db'} = \top$.
- ϕ is $\forall x.\ \psi$. The proof of this case is similar to the $\exists x.\ \psi$ case.

This concludes the proof of the *if* direction.

For the *only if* direction, we assume that for all well-formed assignments ν for ϕ, $[\phi\nu]^{db'} = \top$. We now prove, by structural induction on ϕ, that $[wp(\phi, T \oplus \overline{v}(m))\nu]^{db} = \top$. There are two base cases:

- ϕ is $R(\overline{x})$. If $R \neq T$, then $wp(\phi, T \oplus \overline{v}(m)) = \phi$ and the claim trivially holds since $db(R) = db'(R)$. If $R = T$, then $[\phi\nu]^{db'} = \top$. From this and $db'(T) = db(T) \cup \{\overline{v}(m)\}$, it follows that $\nu(\overline{x}) \in db(T)$ or $\nu(\overline{x}) = \overline{v}(m)$. From this, $[(T(\overline{x}) \vee \overline{x} = \overline{v}(m))\nu]^{db} = \top$. Hence, $[wp(\phi, T \oplus \overline{v}(m))\nu]^{db} = \top$.
- ϕ is $x_1 = x_2$. Then, $wp(\phi, T \oplus \overline{v}(m)) = \phi$ and the claim trivially holds.
- ϕ is $x_1 = c$. Then, $wp(\phi, T \oplus \overline{v}(m)) = \phi$ and the claim trivially holds.

For the induction step, we assume that the claim holds for all sub-formulae of ϕ and we show that it holds for ϕ as well. There are several cases:

- ϕ is $\psi \wedge \gamma$. From $[\phi\nu]^{db'} = \top$, it follows $[\psi\nu]^{db'} = \top$ and $[\gamma\nu]^{db'} = \top$. From this and the induction hypothesis, $[wp(\psi, T \oplus \overline{v}(m))\nu]^{db} = \top$ and $[wp(\gamma, T \oplus \overline{v}(m))\nu]^{db} = \top$. Hence, $[wp(\psi, T \oplus \overline{v}(m))\nu \wedge wp(\gamma, T \oplus \overline{v}(m))\nu]^{db} = \top$ and therefore $[wp(\phi, T \oplus \overline{v}(m))\nu]^{db} = \top$.
- ϕ is $\psi \vee \gamma$. The proof of this case is similar to that of $\psi \wedge \gamma$.
- ϕ is $\neg\psi$. From $[\phi]^{db'} = \top$, it follows that $[\psi]^{db'} = \bot$. From this and the induction hypothesis, it follows that $[wp(\psi, T \oplus \overline{v}(m))\nu]^{db} = \bot$. From this and $wp(\phi, T \oplus \overline{v}(m)) = \neg wp(\psi, T \oplus \overline{v}(m))$, $[wp(\phi, T \oplus \overline{v}(m))\nu]^{db} = \top$.

- ϕ is $\exists x.\ \psi$. From $[\phi\nu]^{db'} = \top$, it follows that there is a value $v \in \mathbf{dom}$ such that $[\psi\nu[x \mapsto v]]^{db'} = \top$. From this, $\nu[x \mapsto v]$ is a well-formed assignment for ψ, and the induction hypothesis, it follows that $[wp(\psi, T \oplus \overline{v}(m))\nu[x \mapsto v]]^{db} = \top$. From this and $wp(\phi, T \oplus \overline{v}(m)) = \exists x.\ wp(\psi, T \oplus \overline{v}(m))$, it follows that $[wp(\phi, T \oplus \overline{v}(m))\nu]^{db} = \top$.
- ϕ is $\forall x.\ \psi$. The proof of this case is similar to the $\exists x.\ \psi$ case.

This completes the proof of the *only if* direction.

Proof of (2). The proof for (2) is similar to that of (1). The only difference is the base case $R(\overline{x})$ in case $R = T$. We show how the proof works for this case. For the *if* direction, assume that $[wp(\phi, T \ominus \overline{v}(m))\nu]^{db} = \top$. From this and $wp(\phi, T \ominus \overline{v}(m)) = T(\overline{x}) \wedge \overline{x} \neq \overline{v}(m)$, it follows that $[(T(\overline{x}) \wedge \overline{x} \neq \overline{v}(m))\nu]^{db} = \top$. From this, $\nu(\overline{x}) \in db(T)$ and $\nu(\overline{x}) \neq \overline{v}(m)$. From this, $\nu(\overline{x}) \in db(T) \setminus \{\overline{v}(m)\}$. Hence, $[T(\overline{x})\nu]^{db'} = \top$.

For the *only if* direction, assume that $[\phi\nu]^{db'} = \top$. From this and $db'(T) = db(T) \setminus \{\overline{v}(m)\}$, it follows that $\nu(\overline{x}) \in db(T)$ and $\nu(\overline{x}) \neq \overline{v}(m)$. From this, $[(T(\overline{x}) \wedge \overline{x} \neq \overline{v}(m))\nu]^{db} = \top$. Hence, $[wp(\phi, T \ominus \overline{v}(m))\nu]^{db} = \top$.

Proof of (3). The third claim immediately follows from (1) and (2). □

Lemma G.2 states that the rules handling the expansion procedure are correct, i.e., that they mimic the operational semantics of the WHILESQL statements of the form $x \leftarrow q$.

Lemma G.2. *Let $m \in Mem$ be a memory, $\langle s, ctx\rangle$ be a runtime state such that $trigger(ctx) = \epsilon$, $u \in UID$ be a user, and Δ be a monitor state. Whenever $\langle \Delta, x \leftarrow q, m, \langle s, ctx\rangle\rangle \overset{\tau'}{\rightsquigarrow}{}^{*}_{u} \langle \Delta', \varepsilon, m', \langle s', ctx'\rangle\rangle$, then $\langle x \leftarrow q, m, \langle s, ctx\rangle\rangle \xrightarrow{\tau''}_{u} \langle \varepsilon, m'', \langle s'', ctx''\rangle\rangle$ and the following conditions hold: (1) $m'(x) = m''(x)$, (2) $s' = s''$, (3) $triggers(ctx') = triggers(ctx'') = \epsilon$, and (4) $\tau' = \tau''$.*

Proof. Let $m \in Mem$ be a memory, $\langle s, ctx\rangle$ be a runtime state such that $trigger(ctx) = \epsilon$, $u \in UID$ be a user, and Δ be a monitor state. Furthermore, we assume that $\langle \Delta, x \leftarrow q, m, \langle s, ctx\rangle\rangle \overset{\tau'}{\rightsquigarrow}{}^{*}_{u} \langle \Delta', \varepsilon, m', \langle s', ctx'\rangle\rangle$. In the computation, we applied the rule F-EXPAND once, the rule F-EXPANDEDCODE multiple times, and the rule F-REMOVEEXPANDEDCODE once. Hence, $\langle \Delta, x \leftarrow q, m, \langle s, ctx\rangle\rangle \overset{\tau'}{\rightsquigarrow}{}^{*}_{u} \langle \Delta', \varepsilon, m', \langle s', ctx'\rangle\rangle$ iff $\langle \Delta, [c], m, \langle s, ctx\rangle\rangle \overset{\tau'}{\rightsquigarrow}{}^{*}_{u} \langle \Delta', [\varepsilon], m', \langle s', ctx'\rangle\rangle$, where $c = expand(\langle s, ctx\rangle, x, q, u)$. This, in turn, happens iff $\langle \Delta, c, m, \langle s, ctx\rangle\rangle \overset{\tau'}{\rightsquigarrow}{}^{*}_{u} \langle \Delta', \varepsilon, m', \langle s', ctx'\rangle\rangle$, where $c = expand(\langle s, ctx\rangle, x, q, u)$. From *expand*'s definition, it follows that $c = decls(\langle s, ctx\rangle, m, u, x \leftarrow q)\ ;\ body(s, m, u, x \leftarrow q)$, where $decls(\langle s, ctx\rangle, m, u, x \leftarrow q)$ is a sequence of `SELECT` queries and $body(s, m, u, x \leftarrow q)$ is a sequence of **if** statements. We claim that (1) at least one of the conditions of the **if** statements in $body(s, m, u, x \leftarrow q)$ is satisfied, and (2) the conditions in the **if** statements in $body(s, m, u, x \leftarrow q)$ are mutually exclusive, i.e., in each execution we execute only one of the **if** statements. Let c' be the **if** statement associated with the satisfied condition. From *expand*'s definition, $c' =$ **if** $cond_{\langle s,ctx\rangle,m,u,x,q}(\overline{t})$ **then** $body_{\langle s,ctx\rangle,m,u,x,q}(\overline{t})$ **else skip**, where $\overline{t}$ is a configuration-consistent execution path. We additionally claim that (3) if $cond_{\langle s,ctx\rangle,m,u,x,q}(\overline{t})$ is satisfied, then the configuration-consistent execution path $\overline{t}$ represents an actual execution of the command and the corresponding triggers, and (4) $body_{\langle s,ctx\rangle,m,u,x,q}(\overline{t})$ correctly implements the semantics of the execution path $\overline{t}$. From (3) and (4), it follows that $\langle x \leftarrow q, m, \langle s, ctx\rangle\rangle \xrightarrow{\tau''}_{u} \langle \varepsilon, m'', \langle s'', ctx''\rangle\rangle$, $m'(x) = m''(x)$, $s' = s''$, $\tau' = \tau''$. Finally, $triggers(ctx') = triggers(ctx'') = \epsilon$, directly follows from the WHILESQL and monitor's semantics.

At least one condition is satisfied. Here we prove our claim that at least one condition in the **if** statements in $body(s, m, u, x \leftarrow q)$ is satisfied. First, observe that there is always at least one configuration-consistent execution path. If $\langle q, \texttt{secEx}\rangle$ is configuration-consistent, then the claim trivially holds as there is an **if** statement with condition $\top$, which is trivially satisfied. Assume now that $\langle q, \texttt{secEx}\rangle$ is not configuration-consistent. We observe that the encoding is such that all possible combinations of variables are covered. Namely, for a query q, there are two **if** statements: one checking whether the integrity constraints are satisfied and one checking whether the constraints are not satisfied. Similarly, for a trigger t, there are four possible **if** statements: one checking if the trigger is enabled and the constraints are satisfied, one checking if the trigger is enabled and the constraints are not satisfied, one checking if the trigger is enabled, and one checking if the trigger is not enabled. From these observations, it follows that there is always at least one satisfied condition.

Mutually exclusive conditions. Here we prove our claim that the conditions in the **if** statements in $body(s, m, u, x \leftarrow q)$ are mutually exclusive. Assume, for contradiction's sake, that this is not the case. This requires that there are two distinct configuration-consistent execution paths $\overline{t}$ and $\overline{t}'$ such that both $cond_{\langle s,ctx\rangle,m,u,x,q}(\overline{t})$ and $cond_{\langle s,ctx\rangle,m,u,x,q}(\overline{t}')$ are satisfied. Since $cond_{\langle s,ctx\rangle,m,u,x,q}(\overline{t})$ and $cond_{\langle s,ctx\rangle,m,u,x,q}(\overline{t}')$ are $\wedge(map(c_{s,m,u,x,q,\overline{t}}, 1 \cdot \ldots \cdot |\overline{t}|))$ and $\wedge(map(c_{s,m,u,x,q,\overline{t}'}, 1 \cdot \ldots \cdot |\overline{t}'|))$, this requires that all $c_{s,m,u,x,q,\overline{t}}(i)$ and $c_{s,m,u,x,q,\overline{t}'}(j)$ are satisfied for $1 \leq i \leq |\overline{t}|$ and $1 \leq j \leq |\overline{t}'|$. Let k be the first position where $\overline{t}$ and $\overline{t}'$ differ. There are two cases:

1) $k = 1$. Then, the two paths differ on the initial query q. There are 6 cases depending on the values for $\overline{t}(1)$ and $\overline{t}'(1)$:
   a) $\overline{t}|_1 = \langle q, \texttt{ok}\rangle$ and $\overline{t}|_1 = \langle q, \texttt{secEx}\rangle$. Since $\overline{t}$ and $\overline{t}'$ are configuration-consistent paths, it follows that $allowed(s, u, q)$ and $\neg allowed(s, u, q)$, leading to a contradiction.
   b) $\overline{t}|_1 = \langle q, \texttt{ok}\rangle$ and $\overline{t}|_1 = \langle q, \texttt{ex}\rangle$. Therefore, $c_{s,m,u,x,q,\overline{t}}(1) = x^{\overline{t}}_{1,\gamma_1} \wedge \ldots \wedge x^{\overline{t}}_{1,\gamma_n}$ and $c_{s,m,u,x,q,\overline{t}'}(1) = \neg(x^{\overline{t}'}_{1,\gamma_1} \wedge \ldots \wedge x^{\overline{t}'}_{1,\gamma_n})$. From *decls*'s definition, it follows that $x^{\overline{t}}_{1,\gamma_i}$ and $x^{\overline{t}'}_{1,\gamma_i}$ are respectively initialized by the statements $x^{\overline{t}}_{1,\gamma_i} \leftarrow$ `SELECT` $wp(\gamma_i, \overline{t}^1)$ and $x^{\overline{t}'}_{1,\gamma_i} \leftarrow$ `SELECT` $wp(\gamma_i, \overline{t}'^1)$. From this, $\overline{t}|_1 = \langle q, \texttt{ok}\rangle$, $\overline{t}|_1 = \langle q, \texttt{ex}\rangle$, and wp's definition, it follows that $wp(\gamma_i, \overline{t}^1) = wp(\gamma_i, \overline{t}'^1)$ for all $\gamma_i \in \Gamma$. This combined with $c_{s,m,u,x,q,\overline{t}}(1) =$

$x^{\overline{t}}_{1,\gamma_1} \wedge \ldots \wedge x^{\overline{t}}_{1,\gamma_n}$ and $c_{s,m,u,x,q,\overline{t}'}(1) = \neg(x^{\overline{t}'}_{1,\gamma_1} \wedge \ldots \wedge x^{\overline{t}'}_{1,\gamma_n})$, leads to a contradiction (since the result of the queries are the same given that in *decls* we just executed `SELECT` queries).

c) $\overline{t}|_1 = \langle q, \mathtt{secEx}\rangle$ and $\overline{t}|_1 = \langle q, \mathtt{ok}\rangle$. Since $\overline{t}$ and $\overline{t}'$ are configuration-consistent paths, it follows that $\neg allowed(s, u, q)$ and $allowed(s, u, q)$, leading to a contradiction.

d) $\overline{t}|_1 = \langle q, \mathtt{secEx}\rangle$ and $\overline{t}|_1 = \langle q, \mathtt{ex}\rangle$. Since $\overline{t}$ and $\overline{t}'$ are configuration-consistent paths, it follows that $\neg allowed(s, u, q)$ and $allowed(s, u, q)$, leading to a contradiction.

e) $\overline{t}|_1 = \langle q, \mathtt{ex}\rangle$ and $\overline{t}|_1 = \langle q, \mathtt{ok}\rangle$. Then, $c_{s,m,u,x,q,\overline{t}}(1) = \neg(x^{\overline{t}}_{1,\gamma_1} \wedge \ldots \wedge x^{\overline{t}}_{1,\gamma_n})$ and $c_{s,m,u,x,q,\overline{t}'}(1) = x^{\overline{t}'}_{1,\gamma_1} \wedge \ldots \wedge x^{\overline{t}'}_{1,\gamma_n}$. From *decls*'s definition, it follows that $x^{\overline{t}}_{1,\gamma_i}$ and $x^{\overline{t}'}_{1,\gamma_i}$ are respectively initialized by the statements $x^{\overline{t}}_{1,\gamma_i} \leftarrow$ `SELECT` $wp(\gamma_i, \overline{t}^{1})$ and $x^{\overline{t}'}_{1,\gamma_i} \leftarrow$ `SELECT` $wp(\gamma_i, \overline{t}'^{1})$. From this, $\overline{t}|_1 = \langle q, \mathtt{ok}\rangle$, $\overline{t}|_1 = \langle q, \mathtt{ex}\rangle$, and *wp*'s definition, it follows that $wp(\gamma_i, \overline{t}^{1}) = wp(\gamma_i, \overline{t}'^{1})$ for all $\gamma_i \in \Gamma$. This combined with $c_{s,m,u,x,q,\overline{t}}(1) = \neg(x^{\overline{t}}_{1,\gamma_1} \wedge \ldots \wedge x^{\overline{t}}_{1,\gamma_n})$ and $c_{s,m,u,x,q,\overline{t}'}(1) = x^{\overline{t}'}_{1,\gamma_1} \wedge \ldots \wedge x^{\overline{t}'}_{1,\gamma_n}$, leads to a contradiction (since the result of the queries are the same given that in *decls* we just executed `SELECT` queries).

f) $\overline{t}|_1 = \langle q, \mathtt{ex}\rangle$ and $\overline{t}|_1 = \langle q, \mathtt{secEx}\rangle$. Since $\overline{t}$ and $\overline{t}'$ are configuration-consistent paths, it follows that $\neg allowed(s, u, q)$ and $allowed(s, u, q)$, leading to a contradiction.

2) $k > 1$. Then the two paths differ on the $(k-1)$-th scheduled trigger t. There are 9 cases depending on the values $\overline{t}(k)$ and $\overline{t}'(k)$:

a) $\overline{t}|_k = \langle t, \mathtt{ok}\rangle$ and $\overline{t}|_k = \langle t, \mathtt{secEx}\rangle$. Since $\overline{t}$ and $\overline{t}'$ are configuration-consistent paths, it follows that $allowed(s, u, t(m))$ and $\neg allowed(s, u, t(m))$, leading to a contradiction.

b) $\overline{t}|_k = \langle t, \mathtt{ok}\rangle$ and $\overline{t}|_k = \langle t, \mathtt{ex}\rangle$. Then, $c_{s,m,u,x,q,\overline{t}}(k) = x^{\overline{t}}_{k,cond} \wedge x^{\overline{t}}_{k,\gamma_1} \wedge \ldots \wedge x^{\overline{t}}_{k,\gamma_n}$ and $c_{s,m,u,x,q,\overline{t}'}(k) = x^{\overline{t}'}_{k,cond} \wedge \neg(x^{\overline{t}'}_{k,\gamma_1} \wedge \ldots \wedge x^{\overline{t}'}_{k,\gamma_n})$. From *decls*'s definition, the variables are initialized as follows: $x^{\overline{t}}_{k,cond} \leftarrow$ `SELECT` $wp(\varphi, \overline{t}^{k-1})$, $x^{\overline{t}'}_{k,cond} \leftarrow$ `SELECT` $wp(\varphi, \overline{t}'^{k-1})$, $x^{\overline{t}}_{k,\gamma_i} \leftarrow$ `SELECT` $wp(\gamma_i, \overline{t}^{k})$, and $x^{\overline{t}'}_{k,\gamma_i} \leftarrow$ `SELECT` $wp(\gamma_i, \overline{t}'^{k})$ for all $\gamma_i \in \Gamma$. From this and $\overline{t}$ and $\overline{t}'$ differ only on the k-th element, it follows that $wp(\varphi, \overline{t}^{k-1}) = wp(\varphi, \overline{t}'^{k-1})$. From $\overline{t}|_k = \langle t, \mathtt{ok}\rangle$ and $\overline{t}|_k = \langle t, \mathtt{ex}\rangle$, it follows that $wp(\gamma_i, \overline{t}^{k}) = wp(\gamma_i, \overline{t}'^{k})$ for all $\gamma_i \in \Gamma$. Hence, all the variables are initialized using the same queries. Since during *decls* we execute only `SELECT` queries, the variables are initialized to the same values. This combined with $c_{s,m,u,x,q,\overline{t}}(k) = x^{\overline{t}}_{k,cond} \wedge x^{\overline{t}}_{k,\gamma_1} \wedge \ldots \wedge x^{\overline{t}}_{k,\gamma_n}$ and $c_{s,m,u,x,q,\overline{t}'}(k) = x^{\overline{t}'}_{k,cond} \wedge \neg(x^{\overline{t}'}_{k,\gamma_1} \wedge \ldots \wedge x^{\overline{t}'}_{k,\gamma_n})$ leads to a contradiction (since just one of the conditions could have been satisfied).

c) $\overline{t}|_k = \langle t, \mathtt{ok}\rangle$ and $\overline{t}|_k = \langle t, \mathtt{dis}\rangle$. Then, $c_{s,m,u,x,q,\overline{t}}(k) = x^{\overline{t}}_{k,cond} \wedge x^{\overline{t}}_{k,\gamma_1} \wedge \ldots \wedge x^{\overline{t}}_{k,\gamma_n}$ and $c_{s,m,u,x,q,\overline{t}'}(k) = \neg x^{\overline{t}'}_{k,cond}$. From *decls*'s definition, the variables are initialized as follows: $x^{\overline{t}}_{k,cond} \leftarrow$ `SELECT` $wp(\varphi, \overline{t}^{k-1})$, and $x^{\overline{t}'}_{k,cond} \leftarrow$ `SELECT` $wp(\varphi, \overline{t}'^{k-1})$. From this and $\overline{t}$ and $\overline{t}'$ differ only on the k-th element, it follows that $wp(\varphi, \overline{t}^{k-1}) = wp(\varphi, \overline{t}'^{k-1})$. Hence, all the variables $x^{\overline{t}}_{k,cond}$ and $x^{\overline{t}'}_{k,cond}$ are initialized using the same queries. Since during *decls* we execute only `SELECT` queries, the variables are initialized to the same values. This combined with $c_{s,m,u,x,q,\overline{t}}(k) = x^{\overline{t}}_{k,cond} \wedge x^{\overline{t}}_{k,\gamma_1} \wedge \ldots \wedge x^{\overline{t}}_{k,\gamma_n}$ and $c_{s,m,u,x,q,\overline{t}'}(k) = \neg x^{\overline{t}'}_{k,cond}$ leads to a contradiction (since just one of the conditions could have been satisfied).

d) $\overline{t}|_k = \langle t, \mathtt{secEx}\rangle$ and $\overline{t}|_k = \langle t, \mathtt{ok}\rangle$. The proof of this case is similar to that of $\overline{t}|_k = \langle t, \mathtt{ok}\rangle$ and $\overline{t}|_k = \langle t, \mathtt{secEx}\rangle$.

e) $\overline{t}|_k = \langle t, \mathtt{secEx}\rangle$ and $\overline{t}|_k = \langle t, \mathtt{ex}\rangle$. Since $\overline{t}$ and $\overline{t}'$ are configuration-consistent paths, it follows that $\neg allowed(s, u, t(m))$ and $allowed(s, u, t(m))$, leading to a contradiction.

f) $\overline{t}|_k = \langle t, \mathtt{secEx}\rangle$ and $\overline{t}|_k = \langle t, \mathtt{dis}\rangle$. Then, $c_{s,m,u,x,q,\overline{t}}(k) = x^{\overline{t}}_{k,cond}$ and $c_{s,m,u,x,q,\overline{t}'}(k) = \neg x^{\overline{t}'}_{k,cond}$. From *decls*'s definition, the variables are initialized as follows: $x^{\overline{t}}_{k,cond} \leftarrow$ `SELECT` $wp(\varphi, \overline{t}^{k-1})$, and $x^{\overline{t}'}_{k,cond} \leftarrow$ `SELECT` $wp(\varphi, \overline{t}'^{k-1})$. From this and $\overline{t}$ and $\overline{t}'$ differ only on the k-th element, it follows that $wp(\varphi, \overline{t}^{k-1}) = wp(\varphi, \overline{t}'^{k-1})$. Hence, all the variables $x^{\overline{t}}_{k,cond}$ and $x^{\overline{t}'}_{k,cond}$ are initialized using the same queries. Since during *decls* we execute only `SELECT` queries, the variables are initialized to the same values. This combined with $c_{s,m,u,x,q,\overline{t}}(k) = x^{\overline{t}}_{k,cond}$ and $c_{s,m,u,x,q,\overline{t}'}(k) = \neg x^{\overline{t}'}_{k,cond}$ leads to a contradiction (since just one of the conditions could have been satisfied).

g) $\overline{t}|_k = \langle t, \mathtt{ex}\rangle$ and $\overline{t}|_k = \langle t, \mathtt{ok}\rangle$. The proof of this case is similar to that of $\overline{t}|_k = \langle t, \mathtt{ex}\rangle$ and $\overline{t}|_k = \langle t, \mathtt{ok}\rangle$.

h) $\overline{t}|_k = \langle t, \mathtt{ex}\rangle$ and $\overline{t}|_k = \langle t, \mathtt{secEx}\rangle$. Since $\overline{t}$ and $\overline{t}'$ are configuration-consistent paths, it follows that $allowed(s, u, t(m))$ and $\neg allowed(s, u, t(m))$, leading to a contradiction.

i) $\overline{t}|_k = \langle t, \mathtt{ex}\rangle$ and $\overline{t}|_k = \langle t, \mathtt{dis}\rangle$. Then, $c_{s,m,u,x,q,\overline{t}}(k) = x^{\overline{t}}_{k,cond} \wedge \neg(x^{\overline{t}}_{k,\gamma_1} \wedge \ldots \wedge x^{\overline{t}}_{k,\gamma_n})$ and $c_{s,m,u,x,q,\overline{t}'}(k) = \neg x^{\overline{t}'}_{k,cond}$. From *decls*'s definition, the variables are initialized as follows: $x^{\overline{t}}_{k,cond} \leftarrow$ `SELECT` $wp(\varphi, \overline{t}^{k-1})$, and $x^{\overline{t}'}_{k,cond} \leftarrow$ `SELECT` $wp(\varphi, \overline{t}'^{k-1})$. From this and $\overline{t}$ and $\overline{t}'$ differ only on the k-th element, it follows that $wp(\varphi, \overline{t}^{k-1}) = wp(\varphi, \overline{t}'^{k-1})$. Hence, all the variables $x^{\overline{t}}_{k,cond}$ and $x^{\overline{t}'}_{k,cond}$ are initialized using the same queries. Since during *decls* we execute only `SELECT` queries, the variables are initialized to the same values. This combined with $c_{s,m,u,x,q,\overline{t}}(k) = x^{\overline{t}}_{k,cond} \wedge \neg(x^{\overline{t}}_{k,\gamma_1} \wedge \ldots \wedge x^{\overline{t}}_{k,\gamma_n})$ and $c_{s,m,u,x,q,\overline{t}'}(k) = \neg x^{\overline{t}'}_{k,cond}$ leads to a contradiction (since just one of the conditions could have been satisfied).

j) $\overline{t}|_k = \langle t, \mathtt{dis}\rangle$ and $\overline{t}|_k = \langle t, \mathtt{ok}\rangle$. The proof of this case is similar to that of $\overline{t}|_k = \langle t, \mathtt{ok}\rangle$ and $\overline{t}|_k = \langle t, \mathtt{dis}\rangle$.

k) $\overline{t}|_k = \langle t, \mathtt{dis}\rangle$ and $\overline{t}|_k = \langle t, \mathtt{secEx}\rangle$. The proof of this case is similar to that of $\overline{t}|_k = \langle t, \mathtt{secEx}\rangle$ and $\overline{t}|_k = \langle t, \mathtt{dis}\rangle$.

1) $\overline{t}|_k = \langle t, \mathtt{dis}\rangle$ and $\overline{t}|_k = \langle t, \mathtt{ex}\rangle$. The proof of this case is similar to that of $\overline{t}|_k = \langle t, \mathtt{ex}\rangle$ and $\overline{t}|_k = \langle t, \mathtt{dis}\rangle$.
Since all cases lead to a contradiction, we proved our claim.

Conditions and execution paths. Here we prove our claim that if $\mathit{cond}_{\langle s,ctx\rangle,m,u,x,q}(\overline{t})$ is satisfied, then the configuration-consistent execution path $\overline{t}$ represents an actual execution of the command and the corresponding triggers. Let $\overline{t}$ be an execution path such that $\mathit{cond}_{\langle s,ctx\rangle,m,u,x,q}(\overline{t})$ is satisfied in the local state $\langle m, s\rangle$. We now show that each prefix of $\overline{t}$ corresponds to a computation on the database (i.e., correspond to a run in the database). We show this by induction on the prefix's length.

For the base case, let $\overline{t}'$ be the prefix of length 1. Then, $\overline{t}'$ is $\langle q, r\rangle$, where $r \in \{\mathtt{secEx}, \mathtt{ok}, \mathtt{ex}\}$.

If $r = \mathtt{secEx}$, it follows that $\mathit{allowed}(s, u, q) = \bot$. From this and *allowed*'s definition, it follows that executing the command on the database throws a security exception.

If $r = \mathtt{ex}$, it follows that $\mathit{allowed}(s, u, q) = \top$. From this and *allowed*'s definition, it follows that executing the command on the database does not throw a security exception. Furthermore, from $r = \mathtt{ex}$, it follows that (1) $\overline{t} = \overline{t}'$, and (2) $\mathit{cond}_{\langle s,ctx\rangle,m,u,x,q}(\overline{t}) = \neg(x^{\overline{t}}_{1,\gamma_1} \wedge \ldots \wedge x^{\overline{t}}_{1,\gamma_n})$. From (1), (2), and $\mathit{cond}_{\langle s,ctx\rangle,m,u,x,q}(\overline{t})$ is satisfied in the local state $\langle m, s\rangle$, it follows that one of $\{x^{\overline{t}}_{1,\gamma_1}, \ldots, x^{\overline{t}}_{1,\gamma_n}\}$ evaluates to $\bot$. From this and *decls*'s definition, there is a γ_i such that the result of SELECT $\mathit{wp}(\gamma_i, \overline{t}^k)$ on the database s is $\bot$. From this and Lemma G.1, executing the query q throws an integrity exception.

Finally, if $r = \mathtt{ok}$, it follows that $\mathit{allowed}(s, u, q) = \top$. From this and *allowed*'s definition, it follows that executing the command on the database does not throw a security exception. Furthermore, it follows that one of the conjuncts in $\mathit{cond}_{\langle s,ctx\rangle,m,u,x,q}(\overline{t})$ is $x^{\overline{t}}_{1,\gamma_1} \wedge \ldots \wedge x^{\overline{t}}_{1,\gamma_n}$. From (1), (2), and $\mathit{cond}_{\langle s,ctx\rangle,m,u,x,q}(\overline{t})$ is satisfied in the local state $\langle m, s\rangle$, it follows that all variables in $\{x^{\overline{t}}_{1,\gamma_1}, \ldots, x^{\overline{t}}_{1,\gamma_n}\}$ evaluates to $\bot$. From this and *decls*'s definition, it follows that for all $\gamma_i \in \Gamma$, the result of SELECT $\mathit{wp}(\gamma_i, \overline{t}^k)$ on the database s is $\top$. From this and Lemma G.1, executing the query q does not throw an integrity exception.

For the induction step, we assume that all prefixes of length less than k correspond to actual computations. We now show that the same holds for prefixes of length k. Then, $\overline{t}' = \overline{t}'' \cdot \langle t, r\rangle$, where $r \in \{\mathtt{secEx}, \mathtt{ok}, \mathtt{ex}, \mathtt{dis}\}$.

If $r = \mathtt{secEx}$, it follows that $\mathit{allowed}(s, u, q) = \bot$. From this and *allowed*'s definition, it follows that executing the trigger on the database throws a security exception. Furthermore, $\mathit{cond}_{\langle s,ctx\rangle,m,u,x,q}(\overline{t})$ contains the conjunct $x^{\overline{t}}_{k,cond}$. From this and $\mathit{cond}_{\langle s,ctx\rangle,m,u,x,q}(\overline{t})$ is satisfied in the local state $\langle m, s\rangle$, it follows that $x^{\overline{t}}_{k,cond}$'s value is $\top$. From this and *decls*'s definition, the result of SELECT $\mathit{wp}(\phi, \overline{t}^{k-1})$ on the database s is $\top$, where ϕ is the trigger's condition. From the induction hypothesis, it follows that $\overline{t}^{k-1}$ represents a computation. From this, the result of SELECT $\mathit{wp}(\psi, \overline{t}^{k-1})$ on the database s is $\top$, and Lemma G.1, it follows that the trigger is enabled in the computation.

If $r = \mathtt{ex}$, it follows that $\mathit{allowed}(s, u, q) = \top$. From this and *allowed*'s definition, it follows that executing the trigger on the database does not throw a security exception. Furthermore, from $r = \mathtt{ex}$, it follows that $\mathit{cond}_{\langle s,ctx\rangle,m,u,x,q}(\overline{t})$ contains the conjunct $x^{\overline{t}}_{k,cond} \wedge \neg(x^{\overline{t}}_{k,\gamma_1} \wedge \ldots \wedge x^{\overline{t}}_{k,\gamma_n})$. From this and $\mathit{cond}_{\langle s,ctx\rangle,m,u,x,q}(\overline{t})$ is satisfied in the local state $\langle m, s\rangle$, it follows that $x^{\overline{t}}_{k,cond}$'s value is $\top$. From this and *decls*'s definition, the result of SELECT $\mathit{wp}(\phi, \overline{t}^{k-1})$ on the database s is $\top$, where ϕ is the trigger's condition. From the induction hypothesis, it follows that $\overline{t}^{k-1}$ represents a computation. From this, the result of SELECT $\mathit{wp}(\psi, \overline{t}^{k-1})$ on the database s is $\top$, and Lemma G.1, it follows that the trigger is enabled in the computation. Furthermore, there is one of $\{x^{\overline{t}}_{1,\gamma_1}, \ldots, x^{\overline{t}}_{1,\gamma_n}\}$ that evaluates to $\bot$. From this and *decls*'s definition, there is a γ_i such that the result of SELECT $\mathit{wp}(\gamma_i, \overline{t}^k)$ on the database s is $\bot$. From this and Lemma G.1, executing the trigger t throws an integrity exception.

If $r = \mathtt{dis}$, $\mathit{cond}_{\langle s,ctx\rangle,m,u,x,q}(\overline{t})$ contains the conjunct $\neg x^{\overline{t}}_{k,cond}$. From this and $\mathit{cond}_{\langle s,ctx\rangle,m,u,x,q}(\overline{t})$ is satisfied in the local state $\langle m, s\rangle$, it follows that $x^{\overline{t}}_{k,cond}$'s value is $\bot$. From this and *decls*'s definition, the result of SELECT $\mathit{wp}(\phi, \overline{t}^{k-1})$ on the database s is $\bot$, where ϕ is the trigger's condition. From the induction hypothesis, it follows that $\overline{t}^{k-1}$ represents a computation. From this, the result of SELECT $\mathit{wp}(\psi, \overline{t}^{k-1})$ on the database s is $\bot$, and Lemma G.1, it follows that the trigger is disabled in the computation.

Finally, if $r = \mathtt{ok}$, it follows that $\mathit{allowed}(s, u, q) = \top$. From this and *allowed*'s definition, it follows that executing the trigger on the database does not throw a security exception. Furthermore, from $r = \mathtt{ex}$, it follows that $\mathit{cond}_{\langle s,ctx\rangle,m,u,x,q}(\overline{t})$ contains the conjunct $x^{\overline{t}}_{k,cond} \wedge x^{\overline{t}}_{k,\gamma_1} \wedge \ldots \wedge x^{\overline{t}}_{k,\gamma_n}$. From this and $\mathit{cond}_{\langle s,ctx\rangle,m,u,x,q}(\overline{t})$ is satisfied in the local state $\langle m, s\rangle$, it follows that the values of $x^{\overline{t}}_{k,cond}, x^{\overline{t}}_{k,\gamma_1}, \ldots, x^{\overline{t}}_{k,\gamma_n}$ are $\top$. From this and *decls*'s definition, the results of SELECT $\mathit{wp}(\phi, \overline{t}^{k-1})$, SELECT $\mathit{wp}(\gamma_1, \overline{t}^k), \ldots,$ SELECT $\mathit{wp}(\gamma_n, \overline{t}^k)$ on the database s are $\top$, where ϕ is the trigger's condition. From the induction hypothesis, it follows that $\overline{t}^{k-1}$ represents a computation. From this, the results of SELECT $\mathit{wp}(\phi, \overline{t}^{k-1})$, SELECT $\mathit{wp}(\gamma_1, \overline{t}^k), \ldots,$ SELECT $\mathit{wp}(\gamma_n, \overline{t}^k)$ on the database s are $\top$, and Lemma G.1, it follows that (1) the trigger t is enabled and (2) executing the trigger t does not throw integrity exceptions.

Encoding and execution paths. Here, we show that $\mathit{body}_{\langle s,ctx\rangle,m,u,x,q}(\overline{t})$ correctly implements the semantics of the execution path $\overline{t}$. Let $\overline{t}$ be a configuration-consistent execution path. If $\overline{t} = \langle q, \mathtt{secEx}\rangle$, then the produced code only assigns $\langle \mathbf{SecEx}, \emptyset\rangle$ to x and does not modify the database. If $\overline{t} = \langle q, \mathtt{ex}\rangle$, then the produced code only assigns $\langle \mathbf{IntEx}, \theta\rangle$ to x, where θ is the set of violated constraints, and does not modify the database. Since we use the weakest precondition, θ contains exactly the violated constraints. If $\overline{t}$ does not ends in ex or secEx and $|\overline{t} = 1|$, the generated code modifies the database state as described in $\overline{t}$, and sets the correct value for x. If $\overline{t}$ does not ends in ex or secEx and $|\bar{t} = 1|$, the generated code modifies the database state as described in $\bar{t}$, and sets the correct value for x. Note also that the generated code produces the correct public observation if the command is a `GRANT`, `REVOKE`, or `CREATE`. If $|\bar{t}| > 1$ and $\bar{t}$ ends in `ex`, then the generated code stores in x the error message $\langle t, \mathbf{IntEx}, \theta\rangle$ (which contains the trigger t that has thrown the exception and the set θ of all violated integrity constraints) and produces the associated database-level event. Finally, if $|\bar{t}| > 1$ and $\bar{t}$ ends in `secEx`, then the generated code stores in x the error message $\langle t, \mathbf{SecEx}, \emptyset\rangle$ (which contains the trigger t that has thrown the exception). ☐

Lemma G.3 states that the local semantics of the security monitor correctly mimics the operational semantics of WHILESQL for all statements that are not queries $x \leftarrow q$.

Lemma G.3. *Let $c \in Com$ be an extended* WHILESQL *program such that $safe(c)$, $m \in Mem$ be a memory, $\langle s, ctx\rangle$ be a runtime state such that $trigger(ctx) = \epsilon$, and $u \in UID$ be a user. Furthermore, let Δ be a monitor state. If $\langle \Delta, c, m, \langle s, ctx\rangle\rangle \overset{\tau'}{\rightsquigarrow}_u \langle \Delta', c', m', \langle s', ctx'\rangle\rangle$, then $\langle strip(c), m, \langle s, ctx\rangle\rangle \xrightarrow{\tau''}{}^{R}_u \langle strip(c'), m'', \langle s'', ctx''\rangle\rangle$ and the following conditions hold: (1) for all variables x occurring in $strip(c)$, then $m'(x) = m''(x)$, (2) $s' = s''$, (3) $triggers(ctx') = triggers(ctx'') = \epsilon$, and (4) $\tau' = \tau''$, where $\rightarrow^R_u$ is the reflexive closure of $\rightarrow_u$.*

Proof. Let $c \in Com$ be an extended WHILESQL program such that $safe(c)$, $m \in Mem$ be a memory, $\langle s, ctx\rangle$ be a runtime state such that $trigger(ctx) = \epsilon$, $u \in UID$ be a user, and Δ be a monitor state. Assume that $\langle \Delta, c, m, \langle s, ctx\rangle\rangle \overset{\tau'}{\rightsquigarrow}_u \langle \Delta', c', m', \langle s', ctx'\rangle\rangle$. We prove our claim by structural induction on the rules defining $\overset{\tau'}{\rightsquigarrow}_u$.

Base case. There are several cases depending on the rule used in the computation:

- Rule F-SKIP. From the rule, it follows that $c = \textbf{skip}$, $\Delta = \Delta'$, $c' = \varepsilon$, $m = m'$, $\langle s, ctx\rangle = \langle s', ctx'\rangle$, and $\tau' = \epsilon$. By applying the E-SKIP rule to $\langle c, m, \langle s, ctx\rangle\rangle$, where $c = \textbf{skip}$, we obtain that $\langle c, m, \langle s, ctx\rangle\rangle \xrightarrow{\tau''}_u \langle c'', m'', \langle s'', ctx''\rangle\rangle$, where $\tau'' = \epsilon$, $c'' = \varepsilon$, $m'' = m$, and $\langle s, ctx\rangle = \langle s'', ctx''\rangle$. Therefore, $c' = c''$, $m' = m''$, $\langle s', ctx'\rangle = \langle s'', ctx''\rangle$, $triggers(ctx') = triggers(ctx'') = \epsilon$, and $\tau' = \tau''$.
- Rule F-ASSIGN. From the rule, it follows that $c = x := e$, $c' = \varepsilon$, $m' = m[x \mapsto [\![e]\!](m)]$, $\langle s', ctx'\rangle = \langle s, ctx\rangle$, and $\tau' = \epsilon$. By applying the E-ASSIGN rule to $\langle c, m, \langle s, ctx\rangle\rangle$, where $c = x := e$, we obtain $\langle c, m, \langle s, ctx\rangle\rangle \xrightarrow{\tau''}_u \langle c'', m'', \langle s'', ctx''\rangle\rangle$, where $\tau'' = \epsilon$, $c'' = \varepsilon$, $m'' = m[x \mapsto [\![e]\!](m)]$, and $\langle s, ctx\rangle = \langle s'', ctx''\rangle$. Therefore, $c' = c''$, $m' = m''$, $\langle s', ctx'\rangle = \langle s'', ctx''\rangle$, $triggers(ctx') = triggers(ctx'') = \epsilon$, and $\tau' = \tau''$.
- Rule F-OUT. From the rule, it follows that $c = \textbf{out}(u', e)$, $c' = \varepsilon$, $m' = m$, $\langle s', ctx'\rangle = \langle s, ctx\rangle$, and $\tau' = \langle u', [\![e]\!](m)\rangle$. By applying the E-OUT rule to $\langle c, m, \langle s, ctx\rangle\rangle$, where $c = \textbf{out}(u', e)$, we obtain $\langle c, m, \langle s, ctx\rangle\rangle \xrightarrow{\tau''}_u \langle c'', m'', \langle s'', ctx''\rangle\rangle$, where $\tau'' = \langle u', [\![e]\!](m)\rangle$, $c'' = \varepsilon$, $m'' = m$, and $\langle s, ctx\rangle = \langle s'', ctx''\rangle$. Therefore, $c' = c''$, $m' = m''$, $\langle s', ctx'\rangle = \langle s'', ctx''\rangle$, $triggers(ctx') = triggers(ctx'') = \epsilon$, and $\tau' = \tau''$.
- Rule F-IFTRUE. From the rule, it follows that $c = \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2$, $[\![e]\!](m) = \top$, $c' = [c_1 \,;\, \textbf{set pc to } \Delta(\mathtt{pc}_u)]$, $m' = m$, $\langle s', ctx'\rangle = \langle s, ctx\rangle$, and $\tau' = \epsilon$. By applying the E-IFTRUE rule to $\langle c, m, \langle s, ctx\rangle\rangle$, where $c = \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2$, we obtain $\langle strip(c), m, \langle s, ctx\rangle\rangle \xrightarrow{\tau''}_u \langle strip(c'), m'', \langle s'', ctx''\rangle\rangle$, where $\tau'' = \epsilon$, $m'' = m$, and $\langle s, ctx\rangle = \langle s'', ctx''\rangle$ (because $[\![e]\!](m) = \top$, $strip(c) = \textbf{if } e \textbf{ then } strip(c_1) \textbf{ else } strip(c_2)$, and $strip(c') = strip(c_1)$). Therefore, $m' = m''$, $\langle s', ctx'\rangle = \langle s'', ctx''\rangle$, $triggers(ctx') = triggers(ctx'') = \epsilon$, and $\tau' = \tau''$.
- Rule F-IFFALSE. The proof of this case is similar to that of the F-IFTRUE case.
- Rule F-WHILETRUE. The proof of this case is similar to that of the F-IFTRUE case.
- Rule F-WHILEFALSE. The proof of this case is similar to that of the F-IFTRUE case.
- Rule F-SEQEMPTY. From the rule, it follows that $c = \varepsilon \,;\, c_1$, $c' = c_1$, $m = m'$, $\langle s, ctx\rangle = \langle s', ctx'\rangle$, and $\tau' = \varepsilon$. By applying the E-SEQEMPTY rule to $\langle c, m, \langle s, ctx\rangle\rangle$, where $c = \varepsilon \,;\, c_1$, we obtain $\langle c, m, \langle s, ctx\rangle\rangle \xrightarrow{\tau''}_u \langle c'', m'', \langle s'', ctx''\rangle\rangle$, where $\tau'' = \epsilon$, $c'' = c_1$, $m'' = m$, and $\langle s, ctx\rangle = \langle s'', ctx''\rangle$. Therefore, $c' = c''$, $m' = m''$, $\langle s', ctx'\rangle = \langle s'', ctx''\rangle$, $triggers(ctx') = triggers(ctx'') = \epsilon$, and $\tau' = \tau''$.
- Rule F-REMOVEEXPANDEDCODE. The proof of this case is similar to that of the F-SEQEMPTY case.
- Rule F-UPDATELABELS. From the rule, it follows that $c = \textbf{set pc to } l$, $c' = \varepsilon$, $m' = m$, $\langle s', ctx'\rangle = \langle s, ctx\rangle$, and $\tau' = \epsilon$. Since the rule modifies only the monitor configuration and $strip(c) = strip(c') = \varepsilon$, it follows that $\langle strip(c), m, \langle s, ctx\rangle\rangle \rightarrow^R_u \langle strip(c'), m'', \langle s'', ctx''\rangle\rangle$, where $m' = m''$, $\langle s', ctx'\rangle = \langle s'', ctx''\rangle$, and $triggers(ctx') = triggers(ctx'') = \epsilon$.

This completes the proof of the base step.

Induction Step. For the induction step, we consider only the F-SEQ and F-EXPANDEDCODE rules.

- Rule F-SEQ. From the rule, it follows that $c = c_1 \,;\, c_2$ and $\langle \Delta, c_1, m, \langle s, ctx\rangle\rangle \overset{\tau'}{\rightsquigarrow}_u \langle \Delta, c_1', m'', \langle s'', ctx''\rangle\rangle$. Furthermore, from $safe(c)$, it follows that we can apply the induction hypothesis. From $\langle \Delta, c_1, m, \langle s, ctx\rangle\rangle \overset{\tau'}{\rightsquigarrow}_u \langle \Delta, c_1', m'', \langle s'', ctx''\rangle\rangle$ and the induction's hypothesis, it follows that $\langle strip(c_1), m, \langle s, ctx\rangle\rangle \xrightarrow{\tau''}{}^{R}_u \langle strip(c_1'), m''', \langle s''', ctx'''\rangle\rangle$ such that m''' and m'' agree on all variables occurring in $strip(c)$, $\tau'' = \tau'$, $s'' = s'''$, and $triggers(ctx'') = triggers(ctx''') = \epsilon$. Observe also that $strip(c_1; c_2) = strip(c_1); strip(c_2)$. There are two cases:
  - the first statement executed in c_1 is of the form $\textbf{set pc to } l$. Therefore, $m''' = m'' = m$, $\langle s, ctx\rangle = \langle s'', ctx''\rangle = \langle s''', ctx'''\rangle$, and $\tau' = \tau'' = \epsilon$. From this, it directly follows that $\langle (strip(c_1) \,;\, strip(c_2)), m, \langle s, ctx\rangle\rangle \xrightarrow{\tau'}{}^{R}_u \langle strip(c_1') \,;\, strip(c_2), m''', \langle s''', ctx'''\rangle\rangle$.

- the first statement executed in c_1 is not of the form **set pc to** l. By applying the E-SEQ rule to $\langle \Delta, \mathit{strip}(c_1 \ ; c_2), m, \langle s, \mathit{ctx}\rangle\rangle$ (given that (1) $\langle \mathit{strip}(c_1), m, \langle s, \mathit{ctx}\rangle\rangle \overset{\tau'}{\rightsquigarrow}_u \langle \mathit{strip}(c_1'), m''', \langle s''', \mathit{ctx}'''\rangle\rangle$, and (2) $\mathit{strip}(c_1 \ ; \ c_2) = \mathit{strip}(c_1) \ ; \ \mathit{strip}(c_2)$) we obtain $\langle(\mathit{strip}(c_1) \ ; \ \mathit{strip}(c_2)), m, \langle s, \mathit{ctx}\rangle\rangle \xrightarrow{\tau'}_u \langle \mathit{strip}(c_1') \ ; \ \mathit{strip}(c_2), m''', \langle s''', \mathit{ctx}'''\rangle\rangle$. Our claim directly follows from (1) m''' and m'' agree on all variables modified in the computation, (2) $\tau'' = \tau'$, (3) $s'' = s'''$, and (4) $\mathit{triggers}(\mathit{ctx}'') = \mathit{triggers}(\mathit{ctx}''') = \epsilon$.

- Rule F-EXPANDEDCODE. From the rule, it follows that $c = [c_1]$ and $\langle \Delta, c_1, m, \langle s, \mathit{ctx}\rangle\rangle \overset{\tau'}{\rightsquigarrow}_u \langle \Delta, c_1', m'', \langle s'', \mathit{ctx}''\rangle\rangle$. Furthermore, from $\mathit{safe}(c)$, it follows that we can apply the induction hypothesis. From $\langle \Delta, c_1, m, \langle s, \mathit{ctx}\rangle\rangle \overset{\tau'}{\rightsquigarrow}_u \langle \Delta, c_1', m'', \langle s'', \mathit{ctx}''\rangle\rangle$ and the induction's hypothesis, it follows that $\langle \mathit{strip}(c_1), m, \langle s, \mathit{ctx}\rangle\rangle \xrightarrow{\tau''}{}^{R}_u \langle \mathit{strip}(c_1'), m''', \langle s''', \mathit{ctx}'''\rangle\rangle$ such that m''' and m'' agree on all variables occurring in $\mathit{strip}(c)$, $\tau'' = \tau'$, $s'' = s'''$, and $\mathit{triggers}(\mathit{ctx}'') = \mathit{triggers}(\mathit{ctx}''') = \epsilon$ (since $\mathit{strip}([c]) = \mathit{strip}(c)$).

This completes the proof of the induction step. $\square$

Finally, Theorem 2 states that the local semantics of the security monitor correctly implements the local semantics of WHILESQL.

Theorem 2. *Let $c \in \mathit{Com}$ be a* WHILESQL *program (without extended commands from §5), $m \in \mathit{Mem}$ be a memory, $\langle s, \mathit{ctx}\rangle$ be a runtime state such that $\mathit{trigger}(\mathit{ctx}) = \epsilon$, and $u \in \mathit{UID}$ be a user. Furthermore, let Δ be a monitor state. If $\langle \Delta, c, m, \langle s, \mathit{ctx}\rangle\rangle \overset{\tau'}{\rightsquigarrow}{}^{*}_u \langle \Delta', \varepsilon, m', \langle s', \mathit{ctx}'\rangle\rangle$, then $\langle c, m, \langle s, \mathit{ctx}\rangle\rangle \xrightarrow{\tau''}{}^{*}_u \langle \varepsilon, m'', \langle s'', \mathit{ctx}''\rangle\rangle$ and the following conditions hold: (1) for all variables x occurring in c, then $m'(x) = m''(x)$, (2) $s' = s''$, (3) $\mathit{triggers}(\mathit{ctx}') = \mathit{triggers}(\mathit{ctx}'') = \epsilon$, and (4) $\tau' = \tau''$.*

Proof. This claim directly follows from Lemma G.2, Lemma G.3. In particular, Lemma G.2 is used to handle statements of the form $x \leftarrow q$, whereas Lemma G.3 is used to handle all other statements. $\square$

G.2. Global semantics

Theorem 3 shows that the monitor's global semantics is transparent for sequential schedulers. We remark, however, that the monitor's global semantics is, in general, not transparent as the monitor modifies the scheduling of commands to avoid timing leaks that may be introduced by the parallel execution of multiple WHILESQL programs.

Theorem 3. *Let $C \in \mathit{Com}^*_{\mathit{UID}}$ be a sequence of* WHILESQL *programs (without the extended commands from §5), $M \in \mathit{Mem}^*_{\mathit{UID}}$ be a sequence of memories, s be a system state, and $\mathcal{S}$ be the sequential scheduler 0^∞. Whenever $\langle \Delta, C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \overset{\tau}{\rightsquigarrow}{}^{*} \langle \Delta', \epsilon, M', \langle s', \mathit{ctx}'\rangle, \mathcal{S}'\rangle$ then $\langle C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\tau}{}^{*} \langle \epsilon, M'', \langle s'', \mathit{ctx}''\rangle, \mathcal{S}''\rangle$ and the following conditions hold: (1) for all $1 \leq i \leq |M|$, for all variables that occur in $C|_i$, then $M'|_i(x) = M''|_i(x)$, (2) $s' = s''$, (3) $\mathit{triggers}(\mathit{ctx}) = \mathit{triggers}(\mathit{ctx}') = \epsilon$, and (4) $\mathcal{S}' = \mathcal{S}''$.*

Proof. The claim directly follows from (1) the use of the sequential scheduler, (2) the application of Theorem 2 to the execution of each program in C, and (3) the fact that the monitor's global semantics does not add new observations to the trace. $\square$

$$deps(\langle u', [\![e]\!](m)\rangle, \langle \Delta, \mathbf{out}(u', e), m, s\rangle \overset{\langle u', [\![e]\!](m)\rangle}{\rightsquigarrow}_u \langle \Delta, \varepsilon, m, s\rangle) = vars(e)$$

$$deps(\langle u, v', o', \tau'\rangle, conf \overset{\langle public, q'\rangle}{\rightsquigarrow}_u conf') = vars(q)$$
$$\text{where } conf = \langle \Delta, \|x \leftarrow q\|, m, s\rangle,\ conf' = \langle \Delta, \varepsilon, m, s\rangle,\ q \text{ is a } \texttt{GRANT}, \texttt{REVOKE}, \text{ or } \texttt{CREATE} \text{ command}$$

$$deps(obs, \langle \Delta, c_1 ; c_2, m, s\rangle \overset{obs}{\rightsquigarrow}_u \langle \Delta', c_1' ; c_2, m', s'\rangle) = deps(obs, \langle \Delta, c_1, m, s\rangle \overset{obs}{\rightsquigarrow}_u \langle \Delta', c_1', m', s'\rangle)$$

$$deps(obs, \langle \Delta, \mathbf{asuser}(u', c), m, s\rangle \overset{obs}{\rightsquigarrow}_u \langle \Delta', c', m', s'\rangle) = deps(obs, \langle \Delta, c, m, s\rangle \overset{obs}{\rightsquigarrow}_{u'} \langle \Delta', c', m', s'\rangle)$$
$$\text{where } query(c) = \top$$

$$deps(obs, \langle \Delta, [c], m, s\rangle \overset{obs}{\rightsquigarrow}_u \langle \Delta', [c'], m', s'\rangle) = deps(obs, \langle \Delta, c, m, s\rangle \overset{obs}{\rightsquigarrow}_u \langle \Delta', c', m', s'\rangle)$$

$$deps(obs, \langle \Delta, C, M, s, n' \cdot \mathcal{S}\rangle \overset{obs}{\rightsquigarrow} \langle \Delta', C', M', s', \mathcal{S}'\rangle) = deps(obs, \langle \Delta, c, m, s\rangle \overset{obs}{\rightsquigarrow}_u \langle \Delta', c', m', s'\rangle)$$
$$\text{where } n = 1 + (n' \ \mathbf{mod}\ |C|), C|_n = \langle u, c\rangle, \text{ and } M|_n = \langle u, m\rangle$$

$$deps(obs, conf \overset{\tau}{\rightsquigarrow} conf') = \emptyset \text{ for any } obs \text{ and } conf \overset{\tau}{\rightsquigarrow} conf' \text{ not matching the above cases}$$

Figure 24: Direct dependencies.

Appendix H. Monitor's soundness

Here, we prove the main result of §5, namely that our enforcement mechanism is sound with respect to our security condition for external attackers. In the following, let $\langle D, \Gamma\rangle$ be a system configuration such that the constraints in Γ are well-formed.

H.1. Auxiliary notation

Let obs be either an observation. We denote by $user(obs)$ the user associated with the observation. Namely, $user(\langle u, o\rangle) = u$ if $u \neq public$ and $user(\langle public, o\rangle) = atk$.

Let obs be either an observation and $conf \overset{obs}{\rightsquigarrow} conf'$ be a step of the local or global semantics. The direct dependencies of obs given $conf \overset{obs}{\rightsquigarrow} conf'$ are defined in Figure 24.

Let c be a WHILESQL extended program. The function $first(c)$ returns the first statement to be executed in c. Formally:

$$first(c) = \begin{cases} first(c_1) & \text{if } \exists c_1.\ c = [c_1] \\ first(c_1) & \text{if } \exists c_1, c_2.\ c = c_1 ; c_2 \\ first(c_1) & \text{if } \exists u, c_1.\ c = \mathbf{asuser}(u, c_1) \\ c & \text{otherwise} \end{cases}$$

H.2. Equivalence definitions

We now introduce a number of equivalence relations that we use throughout the proofs. We first introduce equivalence relations between monitor state, database states, and memories.

Definition 3. Let Δ, Δ' be two monitor states, $\langle m, s\rangle, \langle m', s'\rangle$ be two local states, and u be a user.

We say that Δ and Δ' are L-equivalent, where L is a subset of $Var \cup RC^{pred} \cup \{\mathsf{pc}_u \mid u \in UID\}$, written $\Delta \approx_L \Delta'$, iff for all $x \in L$, $\Delta(x) = \Delta'(x)$.

We say that $s = \langle db, U, S, T, V\rangle$ and $s' = \langle db', U', S', T', V'\rangle$ are configuration equivalent, written $s \equiv^{cfg} s'$, iff $U = U'$, $S = S'$, $T = T'$, and $V = V'$.

We say that $\langle m, s\rangle$ and $\langle m', s'\rangle$ are (V, Q)-equivalent, where $V \subseteq Var$ and $Q \subseteq RC$, written $\langle m, s\rangle \approx_{V,Q} \langle m', s'\rangle$, iff (1) for all $x \in V$, $m(x) = m'(x)$, and (2) for all $q \in Q$, $[q]^{db} = [q]^{db'}$, where $s = \langle db, U, sec, T, V\rangle$ and $s' = \langle db', U', S', T', V'\rangle$.

We now formalize equivalence of local configurations.

Definition 4. Let $\langle \Delta, c, m, \langle s, ctx\rangle\rangle$, $\langle \Delta', c', m', \langle s', ctx'\rangle\rangle$ be two local configurations and $\ell \in \mathcal{L}$ be a label. We say that $\langle \Delta, c, m, \langle s, ctx\rangle\rangle$ and $\langle \Delta', c', m', \langle s', ctx'\rangle\rangle$ are *ℓ-equivalent*, written $\langle \Delta, c, m, \langle s, ctx\rangle\rangle \approx_\ell \langle \Delta', c', m', \langle s', ctx'\rangle\rangle$ iff

- for all $x \in Var \cup \{pc_u \mid u \in UID\}$, $\Delta(x) \sqsubseteq \ell$ iff $\Delta'(x) \sqsubseteq \ell$,
- for all $q \in RC^{pred}$, $\Delta(q) \sqsubseteq \ell$ iff $\Delta'(q) \sqsubseteq \ell$,
- $\langle m, s\rangle \approx_{V,Q} \langle m', s'\rangle$, where $V = \{x \in Var \mid \Delta(x) \sqsubseteq \ell\}$ and $Q = \{q \in RC \mid L_{\mathcal{Q}}(\Delta, q) \sqsubseteq \ell\}$, and
- $\Delta \approx_L \Delta'$, where $L = \{x \in Var \mid \Delta(x) \sqsubseteq \ell\} \cup \{q \in RC^{pred} \mid \Delta(q) \sqsubseteq \ell\} \cup \{\mathsf{pc}_u \mid u \in UID\}$.

Similarly, we say that two global configurations $\langle \Delta, C, M, \langle s, ctx\rangle, \mathcal{S}\rangle$ and $\langle \Delta', C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$ are ℓ-equivalent, written $\langle \Delta, C, M, \langle s, ctx\rangle, \mathcal{S}\rangle \approx_\ell \langle \Delta', C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$, iff $|C| = |C'|$, $|M| = |M'|$, and for all $1 \leq i \leq |C|$, $\langle \Delta, C|_i, M|_i, \langle s, ctx\rangle\rangle \approx_\ell \langle \Delta', C'|_i, M'|_i, \langle s', ctx'\rangle\rangle$

H.3. Results about $\sqcup$

Here we show a simple property of joins in our disclosure lattice, namely that $l_1 \sqcup l_2 \sqsubseteq l_3$ holds iff both $l_1 \sqsubseteq l_3$ and $l_2 \sqsubseteq l_3$ hold. While one of the directions (namely $l_1 \sqcup l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3 \wedge l_2 \sqsubseteq l_3$) holds for disclosure lattices in general, the other one (i.e., $l_1 \sqsubseteq l_3 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqcup l_2 \sqsubseteq l_3$) holds specifically for the determinacy-based lattice. We do not explicitly refer to Proposition H.1 in the rest of the proofs. Observe that from Proposition H.1 it follows that $\bigwedge_i (l_i \sqsubseteq l)$ iff $(\bigsqcup_i l_i) \sqsubseteq l$.

Proposition H.1. *Let D be a database schema, Γ be a set of integrity constraints, $\preceq_{D,\Gamma}^{\twoheadrightarrow}$ be the relation such that $Q \preceq_{D,\Gamma}^{\twoheadrightarrow} Q'$ iff $D, \Gamma \vdash Q' \twoheadrightarrow Q$. Furthermore, the $\preceq_{D,\Gamma}^{\twoheadrightarrow}$-disclosure lattice is $\langle \mathcal{L}, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$, where $\sqsubseteq$ is $\preceq_{D,\Gamma}^{\twoheadrightarrow}$. For any $l_1, l_2, l_3 \in \mathcal{L}$, the following properties hold:*

- *If $l_1 \sqcup l_2 \sqsubseteq l_3$, then $l_1 \sqsubseteq l_3$ and $l_2 \sqsubseteq l_3$.*
- *If $l_1 \sqsubseteq l_3$ and $l_2 \sqsubseteq l_3$, then $l_1 \sqcup l_2 \sqsubseteq l_3$.*

Proof. Let D be a database schema, Γ be a set of integrity constraints, $\preceq_{D,\Gamma}^{\twoheadrightarrow}$ be the relation such that $Q \preceq_{D,\Gamma}^{\twoheadrightarrow} Q'$ iff $D, \Gamma \vdash Q' \twoheadrightarrow Q$. Furthermore, the $\preceq_{D,\Gamma}^{\twoheadrightarrow}$-disclosure lattice is $\langle \mathcal{L}, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$, where $\sqsubseteq$ is $\preceq_{D,\Gamma}^{\twoheadrightarrow}$.

First statement. Let l_1, l_2, l_3 be three elements in $\mathcal{L}$ such that $l_1 \sqcup l_2 \sqsubseteq l_3$. From this and $\mathcal{L}$'s definition, it follows that there are three sets of queries $Q_1, Q_2, Q_3 \in \mathcal{P}RC$ such that $l_i = cl(Q_i)$ for $1 \leq i \leq 3$. From this and $l_1 \sqcup l_2 \sqsubseteq l_3$, it follows that $cl(Q_1) \sqcup cl(Q_2) \sqsubseteq cl(Q_3)$. From this and $\sqcup$'s definition, it follows that $cl(Q_1 \cup Q_2) \sqsubseteq cl(Q_3)$. From the definition of closure, $cl(Q_1) \subseteq cl(Q_1 \cup Q_2)$ and $cl(Q_2) \subseteq cl(Q_1 \cup Q_2)$. From this and the notion of disclosure order, $cl(Q_1) \sqsubseteq cl(Q_1 \cup Q_2)$ and $cl(Q_2) \sqsubseteq cl(Q_1 \cup Q_2)$. From this and $cl(Q_1 \cup Q_2) \sqsubseteq cl(Q_3)$, $cl(Q_1) \sqsubseteq cl(Q_3)$ and $cl(Q_2) \sqsubseteq cl(Q_3)$. Hence, $l_1 \sqsubseteq l_3$ and $l_2 \sqsubseteq l_3$.

Second statement. Let l_1, l_2, l_3 be three elements in $\mathcal{L}$ such that $l_1 \sqsubseteq l_3$ and $l_2 \sqsubseteq l_3$. From this and $\mathcal{L}$'s definition, it follows that there are three sets of queries $Q_1, Q_2, Q_3 \in \mathcal{P}RC$ such that $l_i = cl(Q_i)$ for $1 \leq i \leq 3$. From $cl(Q_1) \sqsubseteq cl(Q_3)$, $cl(Q_2) \sqsubseteq cl(Q_3)$, and property (2) of disclosure lattices, it follows that $Q_1 \preceq_{D,\Gamma}^{\twoheadrightarrow} Q_3$ and $Q_2 \preceq_{D,\Gamma}^{\twoheadrightarrow} Q_3$. From this and $\preceq_{D,\Gamma}^{\twoheadrightarrow}$'s definition, $D, \Gamma \vdash Q_3 \twoheadrightarrow Q_1$ and $D, \Gamma \vdash Q_3 \twoheadrightarrow Q_2$. We claim that $D, \Gamma \vdash Q_3 \twoheadrightarrow Q_1 \cup Q_2$. From this, $Q_1 \cup Q_2 \preceq_{D,\Gamma}^{\twoheadrightarrow} Q_3$. From this and property (2) of disclosure lattices, $cl(Q_1 \cup Q_2) \sqsubseteq cl(Q_3)$. From this and $cl(Q_1) \sqcup cl(Q_2) = cl(Q_1 \cup Q_2)$, $cl(Q_1) \sqcup cl(Q_2) \sqsubseteq cl(Q_3)$.

We now prove our claim that $D, \Gamma \vdash Q_3 \twoheadrightarrow Q_1$ and $D, \Gamma \vdash Q_3 \twoheadrightarrow Q_2$ imply $D, \Gamma \vdash Q_3 \twoheadrightarrow Q_1 \cup Q_2$. Let Q_1, Q_2 and Q_3 be three sets of queries in $\mathcal{P}RC$ such that $D, \Gamma \vdash Q_3 \twoheadrightarrow Q_1$ and $D, \Gamma \vdash Q_3 \twoheadrightarrow Q_2$. Assume, for contradiction's sake, that $D, \Gamma \vdash Q_3 \twoheadrightarrow Q_1 \cup Q_2$ does not hold. From this, it follows that there are two database states db and db' and a query $q' \in Q_1 \cup Q_2$ such that $[q]^{db} = [q]^{db'}$ for all $q \in Q_3$ and $[q']^{db} \neq [q']^{db'}$. If $q' \in Q_1$, then there are two database states db and db' and a query $q' \in Q_1$ such that $[q]^{db} = [q]^{db'}$ for all $q \in Q_3$ and $[q']^{db} \neq [q']^{db'}$. Therefore, $D, \Gamma \vdash Q_3 \twoheadrightarrow Q_1$ does not hold, leading to a contradiction. Similarly, if $q' \in Q_2$, then there are two database states db and db' and a query $q' \in Q_2$ such that $[q]^{db} = [q]^{db'}$ for all $q \in Q_3$ and $[q']^{db} \neq [q']^{db'}$. Therefore, $D, \Gamma \vdash Q_3 \twoheadrightarrow Q_2$ does not hold, leading to a contradiction. Since both cases lead to a contradiction, this completes the proof of our claim. □

H.4. Results about L_Q

Here we state some simple facts about L_Q.

Proposition H.2. *Let Δ and Δ' be two monitor states. If $\Delta(q) \sqsubseteq \ell$ iff $\Delta'(q) \sqsubseteq \ell$ for all $q \in RC^{pred}$, then $L_Q(\Delta, q) \sqsubseteq \ell$ iff $L_Q(\Delta', q) \sqsubseteq \ell$ for all $q \in RC$.*

Proof. Let Δ and Δ' be two monitor states such that $\Delta(q) \sqsubseteq \ell$ iff $\Delta'(q) \sqsubseteq \ell$ for all $q \in RC^{pred}$.

($\Rightarrow$). Assume that $L_Q(\Delta, q) \sqsubseteq \ell$ holds. From this, $\bigsqcup_{Q \in supp_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} \Delta(q') \sqsubseteq \ell$. From this, it follows that $\bigwedge_{Q \in supp_{D,\Gamma}(q)} \bigwedge_{q' \in Q} \Delta(q') \sqsubseteq \ell$. From this and $\Delta(q) \sqsubseteq \ell$ iff $\Delta'(q) \sqsubseteq \ell$ for all $q \in RC^{pred}$, it follows that $\bigwedge_{Q \in supp_{D,\Gamma}(q)} \bigwedge_{q' \in Q} \Delta'(q') \sqsubseteq \ell$. From this, it follows that $\bigsqcup_{Q \in supp_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} \Delta'(q') \sqsubseteq \ell$. From this, $L_Q(\Delta', q) \sqsubseteq \ell$.

($\Leftarrow$). Assume that $L_Q(\Delta', q) \sqsubseteq \ell$ holds. From this, $\bigsqcup_{Q \in supp_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} \Delta'(q') \sqsubseteq \ell$. From this, it follows that $\bigwedge_{Q \in supp_{D,\Gamma}(q)} \bigwedge_{q' \in Q} \Delta'(q') \sqsubseteq \ell$. From this and $\Delta(q) \sqsubseteq \ell$ iff $\Delta'(q) \sqsubseteq \ell$ for all $q \in RC^{pred}$, it follows that $\bigwedge_{Q \in supp_{D,\Gamma}(q)} \bigwedge_{q' \in Q} \Delta(q') \sqsubseteq \ell$. From this, it follows that $\bigsqcup_{Q \in supp_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} \Delta(q') \sqsubseteq \ell$. From this, $L_Q(\Delta, q) \sqsubseteq \ell$. □

Proposition H.3. *Given a monitor state Δ and a predicate query $q \in RC^{pred}$, $\Delta(q) = L_Q(\Delta, q)$.*

Proof. Let Δ be a monitor state and $q \in RC^{pred}$ be a predicate query. From the definition of $L_Q(\Delta, q)$, it follows that $L_Q(\Delta, q) = \bigsqcup_{Q \in supp_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} \Delta(q')$. Since Γ is a set of well-formed integrity constraints and $q \in RC^{pred}$, $supp_{D,\Gamma}(q) = \{\{q\}\}$. From this, $L_Q(\Delta, q) = \Delta(q)$. □

H.5. Results about relaxed NSU checks

We now prove some simple results about relaxed NSU checks.

Proposition H.4. *Let sec_0 be the initial policy, ℓ be a label such that $cl(auth(sec_0, atk)) \sqsubseteq \ell$, and Δ be a monitor state. If $\mathbf{nsu}(x, \mathtt{pc}_u)$ is satisfied for Δ and $\Delta(x) \sqsubseteq \ell$, then $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$.*

Proof. Let sec_0 be the initial policy, ℓ be a label such that $cl(auth(sec_0, atk)) \sqsubseteq \ell$, and Δ be a monitor state. Furthermore, assume that $\mathbf{nsu}(x, \mathtt{pc}_u)$ is satisfied for Δ and $\Delta(x) \sqsubseteq \ell$. From this, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq cl(auth(sec_0, atk)) \vee \Delta(\mathtt{pc}_u) \sqsubseteq \Delta(x)$. There are two cases:

1) $\Delta(\mathtt{pc}_u) \sqsubseteq cl(auth(sec_0, atk))$ holds. From this and $cl(auth(sec_0, atk)) \sqsubseteq \ell$, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$.
2) $\Delta(\mathtt{pc}_u) \sqsubseteq \Delta(x)$ holds. From this and $\Delta(x) \sqsubseteq \ell$, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$.

This completes the proof. □

H.6. Lemmas about the local semantics

Here we present some auxiliary results about the local semantics of our enforcement mechanism.

Lemma H.1 states that whenever the security monitor produces an output, the labels associated with pc and with the event's dependencies are less than (or equal to) the label associated with the user that can observe the event. We remark that this lemma directly implies two facts: (1) observable events for atk occur only in low contexts, i.e., in contexts such that $\Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, atk)$, and (2) all direct flows are authorized, namely atk observes only events that directly depend on information at a lower (or equal) level in the security lattice.

Lemma H.1. *Whenever $r = \langle \Delta, c, m, s\rangle \overset{obs}{\rightsquigarrow}_u \langle \Delta', c', m', s'\rangle$ and $obs \neq \epsilon$, we have $\Delta(deps(obs, r)) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, user(obs))$.*

Proof. Let $\langle \Delta, c, m, s\rangle$ and $\langle \Delta', c', m', s'\rangle$ be local configurations such that $\langle \Delta, c, m, s\rangle \overset{obs}{\rightsquigarrow}_u \langle \Delta', c', m', s'\rangle$ and $obs \neq \epsilon$. In the following, we denote $\langle \Delta, c, m, s\rangle \overset{obs}{\rightsquigarrow}_u \langle \Delta', c', m', s'\rangle$ as r. We now prove, by structural induction on $\overset{\tau}{\rightsquigarrow}_u$, that $\Delta(deps(obs, r)) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, user(obs))$. In the following we consider only on those rules that produce observations. Proving the claim for rules not producing outputs is trivial.

Base Case. There are two cases depending on the rule used to produce the event.

- Rule F-OUT. From the rule, it directly follows that (1) $obs = \langle u', [\![e]\!](m)\rangle$, and (2) $\Delta(e) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, u')$. From $user$'s definition, there are two cases.
  If $u' \neq public$, then $user(\langle u', [\![e]\!](m)\rangle) = u'$. From this and $\Delta(e) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, u')$, it follows that $\Delta(e) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, user(\langle u', [\![e]\!](m)\rangle))$. From this, $deps(\langle u', [\![e]\!](m)\rangle, r) = free(e)$, and $\Delta(free(e)) = \Delta(e)$, it follows that $\Delta(deps(\langle u', [\![e]\!](m)), r\rangle) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, user(\langle u', [\![e]\!](m)\rangle))$. Hence, $\Delta(deps(obs, r)) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, user(obs))$.
  If $u' = public$, then $user(\langle u', [\![e]\!](m)\rangle) = atk$. Moreover, from $L_{\mathcal{U}}$'s definition, $L_{\mathcal{U}}(s, public) = L_{\mathcal{U}}(s, atk)$. From this and $\Delta(e) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, u')$, it follows that $\Delta(e) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, user(\langle u', [\![e]\!](m)\rangle))$. From this, $deps(\langle u', [\![e]\!](m)\rangle, r) = free(e)$, and $\Delta(free(e)) = \Delta(e)$, it follows that $\Delta(deps(\langle u', [\![e]\!](m)), r\rangle) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, user(\langle u', [\![e]\!](m)\rangle))$. Hence, $\Delta(deps(obs, r)) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, user(obs))$.
- Rule F-UPDATECONFIGURATIONOK. From the rule, it follows that (1) $obs = \langle public, q'\rangle$, where $q' = q[v_1 \mapsto [\![v_1]\!](m), \ldots, v_n \mapsto [\![v_n]\!](m)]$ and $vars(q) = \{v_1, \ldots, v_n\}$, (2) $\bigsqcup_{1 \leq i \leq n} \Delta(v_i) \sqsubseteq cl(auth(sec_0, atk))$, and (3) $\Delta(\mathtt{pc}_u) \sqsubseteq cl(auth(sec_0, atk))$. Moreover, $cl(auth(sec_0, atk)) \sqsubseteq L_{\mathcal{U}}(s, atk)$. Hence, $\bigsqcup_{1 \leq i \leq n} \Delta(v_i) \sqsubseteq L_{\mathcal{U}}(s, atk)$ and $\Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, atk)$. From $\bigsqcup_{1 \leq i \leq n} \Delta(v_i) \sqsubseteq L_{\mathcal{U}}(s, atk)$ and $deps$' definition, we have $\Delta(deps(obs, r)) \sqsubseteq L_{\mathcal{U}}(s, atk)$. Hence, $\Delta(deps(obs, r)) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, atk)$. From this and $user(\langle public, q'\rangle) = atk$, $\Delta(deps(obs, r)) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, atk)$. From this and $\langle public, q'\rangle = obs$, $\Delta(deps(obs, r)) \sqcup \Delta(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s, user(obs))$.

This completes the proof of the base case.

Induction Step. The proof for the induction step directly follows from the induction hypothesis (since the rules do not further introduce events). □

Lemma H.2 states that, given a label ℓ, whenever $\mathtt{pc}_u$ becomes high with respect to ℓ, this is caused by a branching statement, a loop statement, or a set-label statement.

Lemma H.2. *Let $\ell \in \mathcal{L}$ be a label. Whenever $\langle \Delta, c, m, \langle s, ctx\rangle\rangle \overset{\tau}{\rightsquigarrow}_u \langle \Delta', c', m', \langle s', ctx'\rangle\rangle$, if $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$ and $\Delta'(\mathtt{pc}_u) \not\sqsubseteq \ell$, then one of the following conditions hold:*

- *$first(c) =$ **if** e **then** c_1 **else** c_2 and $\Delta(e) \not\sqsubseteq \ell$,*
- *$first(c) =$ **while** e **do** c_1 and $\Delta(e) \not\sqsubseteq \ell$, or*
- *$first(c) =$ **set pc to** ℓ_1 and $\ell_1 \not\sqsubseteq \ell$.*

Proof. Let c be a WHILESQL program, $\ell \in \mathcal{L}$ be a label, and $\langle \Delta, c, m, \langle s, ctx\rangle\rangle$ and $\langle \Delta', c', m', \langle s', ctx'\rangle\rangle$ be two local configurations such that (1) $\langle \Delta, c, m, \langle s, ctx\rangle\rangle \overset{\tau}{\rightsquigarrow}_u \langle \Delta', c', m', \langle s', ctx'\rangle\rangle$, and (2) $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$ and $\Delta'(\mathtt{pc}_u) \not\sqsubseteq \ell$. We now prove, by structural induction on $\rightsquigarrow_u$, that our claim holds. In the following, we focus only on the rules that directly modify $\Delta(\mathtt{pc}_u)$. The proof for the other cases is trivial.

Base Case. There are several cases depending on the rule used to derive $\langle \Delta, c, m, \langle s, ctx\rangle\rangle \overset{\mathcal{I}}{\rightsquigarrow}_u \langle \Delta', c', m', \langle s', ctx'\rangle\rangle$.

- Rule F-UPDATELABELS. Then, $\mathit{first}(c) = \textbf{set pc to } \ell_1$. From the rule, it follows that $\Delta'(\mathtt{pc}_u) = \ell_1$. From this and $\Delta'(\mathtt{pc}_u) \not\sqsubseteq \ell$, it follows that $\ell_1 \not\sqsubseteq \ell$.
- Rule F-IFTRUE. Then, $\mathit{first}(c) = \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2$. Furthermore, from $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$, $\Delta'(\mathtt{pc}_u) \not\sqsubseteq \ell$, and $\Delta'(\mathtt{pc}_u) = \Delta(e) \sqcup \Delta(\mathtt{pc}_u)$, it follows that $\Delta(e) \not\sqsubseteq \ell$.
- Rule F-IFFALSE. The proof of this case is similar to that of F-IFTRUE.
- Rule F-WHILETRUE. The proof of this case is similar to that of F-IFTRUE.
- Rule F-WHILEFALSE. The proof of this case is similar to that of F-IFTRUE.

Induction Step. The proof of the induction step directly follows from the induction hypothesis. □

Lemmas H.3 states that, given a label ℓ, whenever we are in a high context (i.e., $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$), there are no changes to the values associated with all variables and queries whose labels are lower than (or equal to) ℓ.

Lemma H.3. *Let sec_0 be the policy used to initialize the monitor and $\ell \in \mathcal{L}$ be a security label such that $cl(auth(sec_0, atk)) \sqsubseteq \ell$. Whenever $\langle \Delta, c, m, \langle s, ctx\rangle\rangle \overset{\mathcal{I}}{\rightsquigarrow}_u \langle \Delta', c', m', \langle s', ctx'\rangle\rangle$, if $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$, then $\langle m, s\rangle \approx_{V,Q} \langle m', s'\rangle$, where $V = \{x \in \mathit{Var} \mid \Delta(x) \sqsubseteq \ell\}$ and $Q = \{q \in \mathit{RC} \mid L_Q(\Delta, q) \sqsubseteq \ell\}$.*

Proof. Let sec_0 be the policy used to initialize the monitor and $\ell \in \mathcal{L}$ be a security label such that $cl(auth(sec_0, atk)) \sqsubseteq \ell$. Furthermore, let $u \in \mathit{UID}$ be a user and $\langle \Delta, c, m, \langle s, ctx\rangle\rangle$, $\langle \Delta', c', m', \langle s', ctx'\rangle\rangle$ be two local configurations such that (1) $\langle \Delta, c, m, \langle s, ctx\rangle\rangle \overset{\mathcal{I}}{\rightsquigarrow}_u \langle \Delta', c', m', \langle s', ctx'\rangle\rangle$, and (2) $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$. Finally, let $V = \{x \in \mathit{Var} \mid \Delta(x) \sqsubseteq \ell\}$ and $Q = \{q \in \mathit{RC} \mid L_Q(\Delta, q) \sqsubseteq \ell\}$. We now show, by structural induction on the rules defining $\rightsquigarrow_u$, that $\langle m, s\rangle \approx_{V,Q} \langle m', s'\rangle$. In the following, we consider only those rules that modify the memory m or the database state s. For the other rules, the claim holds trivially (since $\langle m, s\rangle = \langle m', s'\rangle$).

Base Case. There are several cases depending on the applied rule:

1) Rule F-ASSIGN. Assume, for contradiction's sake, that $\langle m, s\rangle \not\approx_{V,Q} \langle m', s'\rangle$. From the rule, it follows that $s = s'$ and $m' = m[x \mapsto [\![e]\!](m)]$. From this, $\langle m, s\rangle \not\approx_{V,Q} \langle m', s'\rangle$, and $\approx_{V,Q}$'s definition, it follows that (1) $x \in V$ and, therefore, $\Delta(x) \sqsubseteq \ell$, and (2) $m(x) \neq m'(x)$. From the rule, it follows that $\mathbf{nsu}(x, \mathtt{pc}_u)$ holds. From this, $cl(auth(sec_0, atk)) \sqsubseteq \ell$, $\Delta(x) \sqsubseteq \ell$, and Proposition H.4, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This, however, contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
2) Rule F-SELECT. Assume, for contradiction's sake, that $\langle m, s\rangle \not\approx_{V,Q} \langle m', s'\rangle$. From the rule, it follows that $s = s'$ and $m' = m[x \mapsto r]$, where $\langle\langle s, ctx'\rangle, r, \epsilon\rangle = [\![q]\!](s, u)$. From this, $\langle m, s\rangle \not\approx_{V,Q} \langle m', s'\rangle$, and $\approx_{V,Q}$'s definition, it follows that (1) $x \in V$ and, therefore, $\Delta(x) \sqsubseteq \ell$, and (2) $m(x) \neq m'(x)$. From the rule, it follows that $\mathbf{nsu}(x, \mathtt{pc}_u)$. From this, $cl(auth(sec_0, atk)) \sqsubseteq \ell$, $\Delta(x) \sqsubseteq \ell$, and Proposition H.4, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This, however, contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
3) Rule F-UPDATEDATABASEOK. Without loss of generality, we assume that $q = T' \oplus \overline{e}$ and that $\overline{v}$ is the tuple inserted in the database (after evaluating all the expressions). Assume, for contradiction's sake, that $\langle m, s\rangle \not\approx_{V,Q} \langle m', s'\rangle$. From the rule, it follows that $[\![q]\!](s, u) = \langle\langle s', ctx'\rangle, r, \epsilon\rangle$ and $m' = m[x \mapsto r]$. From this, $\langle m, s\rangle \not\approx_{V,Q} \langle m', s'\rangle$, and $\approx_{V,Q}$'s definition, it follows that there are two cases:
   a) $x \in V$ and $m(x) \neq m'(x)$. From $x \in V$ and V's definition, it follows that $\Delta(x) \sqsubseteq \ell$. From the rule, it follows that $\mathbf{nsu}(x, \mathtt{pc}_u)$. From this, $cl(auth(sec_0, atk)) \sqsubseteq \ell$, $\Delta(x) \sqsubseteq \ell$, and Proposition H.4, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This, however, contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
   b) There is a query $q' \in Q$ such that $[q']^s \neq [q']^{s'}$. From $q' \in Q$, $L_Q(\Delta, q') \sqsubseteq \ell$. From this, $\bigsqcup_{Q \in supp_{D,\Gamma}(q')} \bigsqcup_{q'' \in Q} \Delta(q'') \sqsubseteq \ell$. We claim that that for all $Q \in supp_{D,\Gamma}(q')$, $T'(\overline{v}) \in Q$. From this and $\bigsqcup_{Q \in supp_{D,\Gamma}(q')} \bigsqcup_{q'' \in Q} \Delta(q'') \sqsubseteq \ell$, it follows that $\Delta(T'(\overline{v})) \sqsubseteq \ell$. From the rule, it follows that $\mathbf{nsu}(T'(\overline{v}), \mathtt{pc}_u)$. From this, $cl(auth(sec_0, atk)) \sqsubseteq \ell$, $\Delta(T'(\overline{v})) \sqsubseteq \ell$, and Proposition H.4, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This, however, contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
   We now prove our claim that for all $Q \in supp_{D,\Gamma}(q')$, $T'(\overline{v}) \in Q$. Assume that there exists a $Q \in supp_{D,\Gamma}(q')$ such that $T'(\overline{v}) \notin Q$. From $supp_{D,\Gamma}$'s definition, it follows that the predicate queries in Q determine q'. From the database semantics, the result of all queries in Q is the same in s and s' (since we modify only $T'(\overline{v})$). From this and Q determines q', it follows that the result of q' is the same in s and s'. This, however, contradicts $[q']^s \neq [q']^{s'}$.
4) Rule F-UPDATECONFIGURATIONOK. The proof of this case is similar to the F-SELECT case.

Induction Step. The proof for the induction step directly follows from the induction hypothesis (since the rules do not further modify the memory and the database). □

Lemma H.4 states that, given a label ℓ, whenever we are in a high context (i.e., $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$), then (1) there are no changes to the labels associated with variables and queries whose labels are initially below ℓ, and (2) the label of a variable (or query) is initially below ℓ iff it is below ℓ also at the end of the computation.

Lemma H.4. *Let sec_0 be the policy used to initialize the monitor and $\ell \in \mathcal{L}$ be a security label such that $cl(auth(sec_0, atk)) \sqsubseteq \ell$. Whenever $\langle \Delta, c, m, \langle s, ctx\rangle\rangle \overset{\mathcal{I}}{\rightsquigarrow}_u \langle \Delta', c', m', \langle s', ctx'\rangle\rangle$, if $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$, then the following conditions hold:*

- *for all $x \in \mathit{Var}$, $\Delta(x) \sqsubseteq \ell$ iff $\Delta'(x) \sqsubseteq \ell$,*
- *for all $q \in \mathit{RC}^{pred}$, $\Delta(q) \sqsubseteq \ell$ iff $\Delta'(q) \sqsubseteq \ell$,*
- *$\Delta \approx_L \Delta'$, where $L = \{x \in \mathit{Var} \mid \Delta(x) \sqsubseteq \ell\} \cup \{q \in \mathit{RC}^{pred} \mid \Delta(q) \sqsubseteq \ell\}$.*

Proof. Let sec_0 be the policy used to initialize the monitor, $\ell \in \mathcal{L}$ be a security label such that $cl(auth(sec_0, atk)) \sqsubseteq \ell$, $u \in UID$ be a user, and $\langle \Delta, c, m, \langle s, ctx \rangle\rangle, \langle \Delta', c', m', \langle s', ctx' \rangle\rangle$ be two local configurations such that (1) $\langle \Delta, c, m, \langle s, ctx \rangle\rangle \xrightarrow{\mathcal{I}}_u \langle \Delta', c', m', \langle s', ctx' \rangle\rangle$, and (2) $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$. We prove our claim by structural induction on the rules defining $\rightsquigarrow_u$. In the following, we consider only those rules that modify the monitor state Δ for the identifiers in $Var \cup RC^{pred}$. For the other rules, the claim holds trivially.

Base Case. There are several cases depending on the applied rule:

1) Rule F-ASSIGN. From the rule, it follows that (1) $\Delta' = \Delta[x \mapsto \Delta(\mathtt{pc}_u) \sqcup \Delta(e)]$, and (2) $\mathbf{nsu}(x, \mathtt{pc}_u)$. Assume, for contradiction's sake, that our claim does not hold. From $\Delta' = \Delta[x \mapsto \Delta(\mathtt{pc}_u) \sqcup \Delta(e)]$, there are three cases:
   - $\Delta(x) \sqsubseteq \ell$ and $\Delta'(x) \not\sqsubseteq \ell$. From $\Delta(x) \sqsubseteq \ell$, $\mathbf{nsu}(x, \mathtt{pc}_u)$, $cl(auth(sec_0, atk)) \sqsubseteq \ell$, and Proposition H.4, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
   - $\Delta(x) \not\sqsubseteq \ell$ and $\Delta'(x) \sqsubseteq \ell$. From $\Delta'(x) = \Delta(\mathtt{pc}_u) \sqcup \Delta(e)$ and $\Delta'(x) \sqsubseteq \ell$, it follows that $\Delta(\mathtt{pc}_u) \sqcup \Delta(e) \sqsubseteq \ell$. From this, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
   - $\Delta(x) \sqsubseteq \ell$, $\Delta'(x) \sqsubseteq \ell$, and $\Delta(x) \neq \Delta'(x)$. From $\Delta(x) \sqsubseteq \ell$, $\mathbf{nsu}(x, \mathtt{pc}_u)$, $cl(auth(sec_0, atk)) \sqsubseteq \ell$, and Proposition H.4, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
2) Rule F-SELECT. From the rule, it follows that $\Delta' = \Delta[x \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_\varphi]$, where $\ell_\varphi = L_{\mathcal{Q}}(\Delta, \mathtt{SELECT}\ \varphi) \sqcup \bigsqcup_{v \in vars(\varphi)} \Delta(v)$, and $\mathbf{nsu}(x, \mathtt{pc}_u)$. Assume, for contradiction's sake, that our claim does not hold. From $\Delta' = \Delta[x \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_\varphi]$, there are three cases:
   - $\Delta(x) \sqsubseteq \ell$ and $\Delta'(x) \not\sqsubseteq \ell$. From $\Delta(x) \sqsubseteq \ell$, $\mathbf{nsu}(x, \mathtt{pc}_u)$, $cl(auth(sec_0, atk)) \sqsubseteq \ell$, and Proposition H.4, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
   - $\Delta(x) \not\sqsubseteq \ell$ and $\Delta'(x) \sqsubseteq \ell$. From $\Delta'(x) = \Delta(\mathtt{pc}_u) \sqcup \ell_\varphi$ and $\Delta'(x) \sqsubseteq \ell$, it follows that $\Delta(\mathtt{pc}_u) \sqcup \ell_\varphi \sqsubseteq \ell$. From this, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
   - $\Delta(x) \sqsubseteq \ell$, $\Delta'(x) \sqsubseteq \ell$, and $\Delta(x) \neq \Delta'(x)$. From $\Delta(x) \sqsubseteq \ell$, $\mathbf{nsu}(x, \mathtt{pc}_u)$, $cl(auth(sec_0, atk)) \sqsubseteq \ell$, and Proposition H.4, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
3) Rule F-UPDATEDATABASEOK. From the rule, it follows that (1) $\Delta' = \Delta[T(\overline{v}) \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_e, x \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_e]$, where $\ell_e = \bigsqcup_{1 \leq i \leq |T|} \Delta(e_i)$, (2) $\mathbf{nsu}(T(\overline{v}), \mathtt{pc}_u)$, and (3) $\mathbf{nsu}(x, \mathtt{pc}_u)$. Assume, for contradiction's sake, that our claim does not hold. From $\Delta' = \Delta[T(\overline{v}) \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_e, x \mapsto \Delta(\mathtt{pc}_u) \sqcup \ell_e]$, there are six cases:
   - $\Delta(x) \sqsubseteq \ell$ and $\Delta'(x) \not\sqsubseteq \ell$. From $\Delta(x) \sqsubseteq \ell$, $\mathbf{nsu}(x, \mathtt{pc}_u)$, $cl(auth(sec_0, atk)) \sqsubseteq \ell$, and Proposition H.4, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
   - $\Delta(x) \not\sqsubseteq \ell$ and $\Delta'(x) \sqsubseteq \ell$. From $\Delta'(x) = \Delta(\mathtt{pc}_u) \sqcup \ell_e$ and $\Delta'(x) \sqsubseteq \ell$, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
   - $\Delta(x) \sqsubseteq \ell$, $\Delta'(x) \sqsubseteq \ell$, and $\Delta(x) \neq \Delta'(x)$. From $\Delta(x) \sqsubseteq \ell$, $\mathbf{nsu}(x, \mathtt{pc}_u)$, $cl(auth(sec_0, atk)) \sqsubseteq \ell$, and Proposition H.4, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
   - $\Delta(T(\overline{v})) \sqsubseteq \ell$ and $\Delta'(T(\overline{v})) \not\sqsubseteq \ell$. From the rule, $\mathbf{nsu}(T(\overline{v}), \mathtt{pc}_u)$. From this, $\Delta(T(\overline{v})) \sqsubseteq \ell$, $cl(auth(sec_0, atk)) \sqsubseteq \ell$, and Proposition H.4, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
   - $\Delta(T(\overline{v})) \not\sqsubseteq \ell$ and $\Delta'(T(\overline{v})) \sqsubseteq \ell$. From $\Delta'(T(\overline{v})) = \Delta(\mathtt{pc}_u) \sqcup \ell_e$ and $\Delta'(T(\overline{v})) \sqsubseteq \ell$, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
   - $\Delta(T(\overline{v})) \sqsubseteq \ell$, $\Delta'(T(\overline{v})) \sqsubseteq \ell$, and $\Delta(T(\overline{v})) \neq \Delta'(T(\overline{v}))$. From $\Delta'(T(\overline{v})) \sqsubseteq \ell$ and $\Delta'(T(\overline{v})) = \Delta(\mathtt{pc}_u) \sqcup \ell_e$, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$. This contradicts our assumption that $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.
4) Rule F-UPDATECONFIGURATION-OK. The proof of this case is similar to that of F-SELECT.

Induction Step. The proof for the induction step directly follows from the induction hypothesis (since the rules do not further modify the memory and the database). □

Lemma H.5 states that whenever $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$ and $cl(auth(sec_0, atk)) \sqsubseteq \ell$ then there are no changes to the database configuration.

Lemma H.5. *Let sec_0 be the policy used to initialize the monitor and $\ell \in \mathcal{L}$ be a security label. Whenever $\langle \Delta, c, m, \langle s, ctx \rangle\rangle \xrightarrow{\mathcal{I}}_u \langle \Delta', c', m', \langle s', ctx' \rangle\rangle$, if $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$ and $cl(auth(sec_0, atk)) \sqsubseteq \ell$, then $s \equiv^{cfg} s'$.*

Proof. Let sec_0 be the policy used to initialize the monitor and $\ell \in \mathcal{L}$ be a security label. Furthermore, let $u \in UID$ be a user and $\langle \Delta, c, m, \langle s, ctx \rangle\rangle, \langle \Delta', c', m', \langle s', ctx' \rangle\rangle$ be two local configurations such that (1) $\langle \Delta, c, m, \langle s, ctx \rangle\rangle \xrightarrow{\mathcal{I}}_u \langle \Delta', c', m', \langle s', ctx' \rangle\rangle$, (2) $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$, and (3) $cl(auth(sec_0, atk)) \sqsubseteq \ell$. We now show, by structural induction on the rules defining $\rightsquigarrow_u$, that $s \equiv^{cfg} s'$. In the following, we consider only those rules that modify the database configuration. For the other rules, the claim holds trivially (since the configuration is the same in s and s').

Base Case. There only interesting case is the rule F-UPDATECONFIGURATION-OK. From the rule, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq cl(auth(sec_0, atk))$. From this and $cl(auth(sec_0, atk)) \sqsubseteq \ell$, it follows that $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$, leading to a contradiction.

Induction Step. The proof of the induction step follows from the induction hypothesis. □

Lemma H.6 states that, under appropriate conditions, executing the same command on two ℓ-equivalent states produces outputs that are indistinguishable for the attacker atk.

Lemma H.6. *Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle\rangle$, $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1' \rangle\rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle\rangle$, and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2' \rangle\rangle$ be four local configurations, and $\ell \in \mathcal{L}$ be a label. If the following conditions hold:*

1) $s_1 \equiv^{cfg} s_2$,
2) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \approx_\ell \langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$,
3) $L_{\mathcal{U}}(s_1, atk) \sqsubseteq \ell$ *and* $L_{\mathcal{U}}(s_2, atk) \sqsubseteq \ell$,
4) $c_1 = c_2$,
5) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$,
6) $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$,
then $\tau_1\restriction_{atk} = \tau_2\restriction_{atk}$.

Proof. Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle$, $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$, and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ be four local configurations, and $\ell \in \mathcal{L}$ be a label such that the following conditions hold: (1) $s_1 \equiv^{cfg} s_2$, (2) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \approx_\ell \langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$, (3) $L_{\mathcal{U}}(s_1, atk) \sqsubseteq \ell$ and $L_{\mathcal{U}}(s_2, atk) \sqsubseteq \ell$, (4) $c_1 = c_2$, (5) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$, (6) $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$, We prove our claim by induction on the rules defining $\rightsquigarrow_u$. Without loss of generality, we focus only on the rules producing observations. The claim trivially holds for all rules that do not produce observations.

Base Case. There are a number of cases depending on the rule applied to derive $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$.

- Rule F-OUT. From the rule, it follows that $c_1 = \textbf{out}(u', e)$. From this and (4), it follows that $c_2 = \textbf{out}(u', e)$. In the following, we assume that $u' = atk$. If $u' = public$, then the proof is identical, whereas if $u' \notin \{atk, public\}$, then the proof is trivial. From $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$ and $c_1 = \textbf{out}(atk, e)$, it follows that $\Delta_1 = \Delta_1'$, $m_1 = m_1'$, $\tau_1 = \langle u', [\![e]\!](m_1)\rangle$, $c_1' = \varepsilon$, and $\langle s_1, ctx_1\rangle = \langle s_1', ctx_1'\rangle$. From $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$, $c_2 = \textbf{out}(u', e)$, and the F-OUT rule, it follows that $\Delta_2 = \Delta_2'$, $m_2 = m_2'$, $\tau_2 = \langle u', [\![e]\!](m_2)\rangle$, $c_2' = \varepsilon$, and $\langle s_2, ctx_2\rangle = \langle s_2', ctx_2'\rangle$. From the rule, it also follows that $\Delta_1(e) \sqcup \Delta_1(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s_1, atk)$ and $\Delta_2(e) \sqcup \Delta_2(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s_2, atk)$. From this, it follows that $\bigwedge_{y \in free(e)} \Delta_1(y) \sqsubseteq L_{\mathcal{U}}(s_1, atk)$ and $\bigwedge_{y \in free(e)} \Delta_2(y) \sqsubseteq L_{\mathcal{U}}(s_2, atk)$. From this, (2), and (3), it follows that $\bigwedge_{y \in free(e)} m_1(y) = m_2(y)$. From this, it follows that $[\![e]\!](m_1) = [\![e]\!](m_2)$. From this, $\tau_1 = \langle u', [\![e]\!](m_1)\rangle$, and $\tau_2 = \langle u', [\![e]\!](m_2)\rangle$, it follows that $\tau_1 = \tau_2$. Therefore, $\tau_1\restriction_{atk} = \tau_2\restriction_{atk}$.
- Rule F-UPDATECONFIGURATIONOK. From the rule, it follows that $c_1 = \|x \leftarrow q\|$. From this and (4), it follows that $c_2 = \|x \leftarrow q\|$. From $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$ and $c_1 = \|x \leftarrow q\|$, it follows that $\Delta_1' = \Delta_1[x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in vars(q)} \Delta_1(v)]$, $m_1' = m_1[x \mapsto r']$, $\tau_1 = \langle public, q[v_1 \mapsto [\![v_1]\!](m_1), \ldots, v_n \mapsto [\![v_n]\!](m_1)]\rangle$, $c_1' = \varepsilon$, and $\langle s_1, ctx_1\rangle = \langle s_1', ctx_1'\rangle$. From $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$, $c_2 = \textbf{out}(u', e)$, and the F-UPDATECONFIGURATIONOK rule, it follows that $\Delta_2' = \Delta_2[x \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in vars(q)} \Delta_2(v)]$, $m_2' = m_2[x \mapsto r'']$, $\tau_2 = \langle public, q[v_1 \mapsto [\![v_1]\!](m_2), \ldots, v_n \mapsto [\![v_n]\!](m_2)]\rangle$, $c_2' = \varepsilon$, and $\langle s_2, ctx_2\rangle = \langle s_2', ctx_2'\rangle$. From the rule, it also follows that $\bigsqcup_{v \in vars(q)} \Delta_1(v) \sqsubseteq cl(auth(sec_0, atk))$ and $\bigsqcup_{v \in vars(q)} \Delta_2(v) \sqsubseteq cl(auth(sec_0, atk))$. Hence, $cl(auth(sec_0, atk)) \sqsubseteq L_{\mathcal{U}}(s_1, atk)$, and $cl(auth(sec_0, atk)) \sqsubseteq L_{\mathcal{U}}(s_2, atk)$. From this, (2), and (3), it follows that $\bigwedge_{v \in vars(q)} m_1(v) = m_2(v)$. From this, $q[v_1 \mapsto [\![v_1]\!](m_1), \ldots, v_n \mapsto [\![v_n]\!](m_1)] = q[v_1 \mapsto [\![v_1]\!](m_2), \ldots, v_n \mapsto [\![v_n]\!](m_2)]$. Hence, $\tau_1 = \tau_2$. Therefore, $\tau_1\restriction_{atk} = \tau_2\restriction_{atk}$.

Induction Step. The proof of the induction step directly follows from the induction hypothesis. □

Lemma H.7 states that, under appropriate conditions, executing the same command on two ℓ-equivalent states modifies the database configuration in the same way.

Lemma H.7. *Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle$, $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$, and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ be four local configurations, and $\ell \in \mathcal{L}$ be a label. If the following conditions hold:*
1) $s_1 \equiv^{cfg} s_2$,
2) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \approx_\ell \langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$,
3) $c_1 = c_2$,
4) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$,
5) $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$,
6) $cl(auth(sec_0, atk)) \sqsubseteq \ell$,
then $s_1' \equiv^{cfg} s_2'$.

Proof. Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle$, $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$, and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ be four local configurations, and $\ell \in \mathcal{L}$ be a label such that the following conditions hold: 1) $s_1 \equiv^{cfg} s_2$, 2) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \approx_\ell \langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$, 3) $c_1 = c_2$, 4) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$, 5) $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$, 6) $cl(auth(sec_0, atk)) \sqsubseteq \ell$. We prove our claim by induction on the rules defining $\rightsquigarrow_u$. In the following, we focus only on rules that modify the database configuration. For rules that do not modify the database configuration, the claim directly follows from $s_1 \equiv^{cfg} s_2$.

Base Case. The only interesting case is the rule F-UPDATECONFIGURATIONOK. From the rule, it follows that $c_1 = \|x \leftarrow q\|$, where q is a configuration command. From this and (3), it follows that $c_2 = \|x \leftarrow q\|$. From $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$ and $c_1 = \|x \leftarrow q\|$, it follows that $\Delta_1' = \Delta_1[x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in vars(q)} \Delta_1(v)]$,

$m_1' = m_1[x \mapsto r_1]$, $vars(q) = \{v_1, \ldots, v_n\}$, $q_1' = q[v_1 \mapsto [\![v_1]\!](m_1), \ldots, v_n \mapsto [\![v_n]\!](m)]$, $\tau_1 = \langle public, q_1' \rangle$, $c_1' = \varepsilon$, and $[\![q_1']\!](\langle s_1, ctx_1 \rangle) = \langle \langle s_1', ctx_1' \rangle, r_1, \epsilon \rangle$. Similarly, from $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle \rangle \xrightarrow{\tau_2}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2' \rangle \rangle$ and $c_2 = \|x \leftarrow q\|$, it follows that $\Delta_2' = \Delta_2[x \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in vars(q)} \Delta_2(v)]$, $m_2' = m_2[x \mapsto r_2]$, $vars(q) = \{v_1, \ldots, v_n\}$, $q_2' = q[v_1 \mapsto [\![v_1]\!](m_1), \ldots, v_n \mapsto [\![v_n]\!](m)]$, $\tau_2 = \langle public, q_2' \rangle$, $c_2' = \varepsilon$, and $[\![q_2']\!](\langle s_2, ctx_2 \rangle) = \langle \langle s_2', ctx_2' \rangle, r_2, \epsilon \rangle$. From the rule, it follows that $\bigsqcup_{v \in vars(q)} \Delta_1(v) \sqsubseteq cl(auth(sec_0, atk))$. From this, it follows that $\bigwedge_{v \in vars(q)} \Delta_1(v) \sqsubseteq cl(auth(sec_0, atk))$. From this and (6), it follows that $\bigwedge_{v \in vars(q)} \Delta_1(v) \sqsubseteq \ell$. From this and (2), it follows that $\bigwedge_{v \in vars(q)} m_1(v) = m_2(v)$. From this, $q_1' = q_2'$. From this, (1), $[\![q_1']\!](\langle s_1, ctx_1 \rangle) = \langle \langle s_1', ctx_1' \rangle, r_1, \epsilon \rangle$, and $[\![q_2']\!](\langle s_2, ctx_2 \rangle) = \langle \langle s_2', ctx_2' \rangle, r_2, \epsilon \rangle$, it directly follows that $s_1' \equiv^{cfg} s_2'$ (since the initial configuration is the same and the database semantics is deterministic).

Induction Step. The proof of the induction step directly follows from the induction hypothesis. ☐

Lemma H.8 states that, given a label ℓ, whenever we are in a low context (i.e., $\Delta(\mathtt{pc}_u) \sqsubseteq \ell$), then executing the same command on two ℓ-equivalent states produces to ℓ-equivalent states.

Lemma H.8. *Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle \rangle$, $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1' \rangle \rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle \rangle$, and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2' \rangle \rangle$ be four local configurations, and $\ell \in \mathcal{L}$ be a label. If the following conditions hold:*
1) $s_1 \equiv^{cfg} s_2$,
2) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle \rangle \approx_\ell \langle \Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle \rangle$,
3) $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and $\Delta_2(\mathtt{pc}_u) \sqsubseteq \ell$,
4) $c_1 = c_2$,
5) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle \rangle \xrightarrow{\tau_1}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1' \rangle \rangle$,
6) $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle \rangle \xrightarrow{\tau_2}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2' \rangle \rangle$,
then $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1' \rangle \rangle \approx_\ell \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2' \rangle \rangle$.

Proof. Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle \rangle$, $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1' \rangle \rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle \rangle$, and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2' \rangle \rangle$ be four local configurations, and $\ell \in \mathcal{L}$ be a label such that the following conditions hold:
1) $s_1 \equiv^{cfg} s_2$,
2) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle \rangle \approx_\ell \langle \Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle \rangle$,
3) $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and $\Delta_2(\mathtt{pc}_u) \sqsubseteq \ell$,
4) $c_1 = c_2$,
5) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle \rangle \xrightarrow{\tau_1}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1' \rangle \rangle$,
6) $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle \rangle \xrightarrow{\tau_2}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2' \rangle \rangle$.
We prove our claim by induction on the rules defining $\rightsquigarrow_u$. In the following, we focus only on those rules that modify the monitor state, the database, or the memory. For the other rules, the claim directly follows from (2).

Base Case. There are a number of cases depending on the rule applied to derive $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle \rangle \xrightarrow{\tau_1}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1' \rangle \rangle$.

- Rule F-UPDATELABELS. From the rule, it follows that $c_1 =$ **set pc to** l. From this and (4), it follows that $c_2 =$ **set pc to** l. From $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle \rangle \xrightarrow{\tau_1}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1' \rangle \rangle$ and $c_1 =$ **set pc to** l, it follows that $\Delta_1' = \Delta_1[\mathtt{pc}_u \mapsto l]$, $m_1 = m_1'$, $\tau_1 = \epsilon$, $c_1' = \varepsilon$, and $\langle s_1, ctx_1 \rangle = \langle s_1', ctx_1' \rangle$. Similarly, from $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle \rangle \xrightarrow{\tau_2}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2' \rangle \rangle$ and $c_2 =$ **set pc to** l, it follows that $\Delta_2' = \Delta_2[\mathtt{pc}_u \mapsto l]$, $m_2 = m_2'$, $\tau_2 = \epsilon$, $c_2' = \varepsilon$, and $\langle s_2, ctx_2 \rangle = \langle s_2', ctx_2' \rangle$. Assume, for contradiction's sake, that $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1' \rangle \rangle \not\approx_\ell \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2' \rangle \rangle$. From this, $m_1 = m_1'$, $m_2 = m_2'$, $\langle s_1, ctx_1 \rangle = \langle s_1', ctx_1' \rangle$, $\langle s_2, ctx_2 \rangle = \langle s_2', ctx_2' \rangle$, and (2), there are two cases:
  - $\Delta_1'(\mathtt{pc}_u) \sqsubseteq \ell$ and $\Delta_2'(\mathtt{pc}_u) \not\sqsubseteq \ell$ (or vice versa). From $\Delta_1' = \Delta_1[\mathtt{pc}_u \mapsto l]$ and $\Delta_2' = \Delta_2[\mathtt{pc}_u \mapsto l]$, it follows that $\Delta_1'(\mathtt{pc}_u) = \Delta_2'(\mathtt{pc}_u)$. From this and $\Delta_1'(\mathtt{pc}_u) \sqsubseteq \ell$, it follows that $\Delta_2'(\mathtt{pc}_u) \sqsubseteq \ell$, leading to a contradiction.
  - $\Delta_1'(\mathtt{pc}_u) \sqsubseteq \ell$, $\Delta_2'(\mathtt{pc}_u) \sqsubseteq \ell$, and $\Delta_1'(\mathtt{pc}_u) \neq \Delta_2'(\mathtt{pc}_u)$. From $\Delta_1' = \Delta_1[\mathtt{pc}_u \mapsto l]$ and $\Delta_2' = \Delta_2[\mathtt{pc}_u \mapsto l]$, it follows that $\Delta_1'(\mathtt{pc}_u) = \Delta_2'(\mathtt{pc}_u)$, leading to a contradiction.
- Rule F-ASSIGN. From the rule, it follows that $c_1 = x := e$. From this and (4), it follows that $c_2 = x := e$. From $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle \rangle \xrightarrow{\tau_1}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1' \rangle \rangle$ and $c_1 = x := e$, it follows that $\Delta_1' = \Delta_1[x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e)]$, $m_1' = m_1[x \mapsto [\![e]\!](m_1)]$, $\tau_1 = \epsilon$, $c_1' = \varepsilon$, and $\langle s_1, ctx_1 \rangle = \langle s_1', ctx_1' \rangle$. Similarly, from $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle \rangle \xrightarrow{\tau_2}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2' \rangle \rangle$ and $c_2 = x := e$, it follows that $\Delta_2' = \Delta_2[x \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \Delta_2(e)]$, $m_2' = m_2[x \mapsto [\![e]\!](m_2)]$, $\tau_2 = \epsilon$, $c_2' = \varepsilon$, and $\langle s_2, ctx_2 \rangle = \langle s_2', ctx_2' \rangle$. Assume, for contradiction's sake, that $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1' \rangle \rangle \not\approx_\ell \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2' \rangle \rangle$. From this, $\langle s_1, ctx_1 \rangle = \langle s_1', ctx_1' \rangle$, $\langle s_2, ctx_2 \rangle = \langle s_2', ctx_2' \rangle$, and (2), there are three cases:
  - $\Delta_1'(x) \sqsubseteq \ell$ and $\Delta_2'(x) \not\sqsubseteq \ell$ (or vice versa). From the rule, it follows that $\Delta_1' = \Delta_1[x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e)]$ and $\Delta_2' = \Delta_2[x \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \Delta_2(e)]$. From this and $\Delta_1'(x) \sqsubseteq \ell$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e) \sqsubseteq \ell$. From this, $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and $\bigwedge_{y \in free(e)} \Delta_1(y) \sqsubseteq \ell$. From $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and (2), it follows that $\Delta_1(\mathtt{pc}_u) = \Delta_2(\mathtt{pc}_u)$. From $\bigwedge_{y \in free(e)} \Delta_1(y) \sqsubseteq \ell$ and (2), it follows that $\bigwedge_{y \in free(e)} \Delta_1(y) = \Delta_2(y)$. From $\Delta_1(\mathtt{pc}_u) = \Delta_2(\mathtt{pc}_u)$ and $\bigwedge_{y \in free(e)} \Delta_1(y) = \Delta_2(y)$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e) = \Delta_2(\mathtt{pc}_u) \sqcup \Delta_2(e)$. From this, $\Delta_1' = \Delta_1[x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e)]$, and $\Delta_2' = \Delta_2[x \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \Delta_2(e)]$, it follows that $\Delta_2'(x) = \Delta_1'(x)$. From this and $\Delta_1'(x) \sqsubseteq \ell$, it follows that $\Delta_2'(x) \sqsubseteq \ell$, leading to a contradiction.

- $\Delta'_1(x) \sqsubseteq \ell$, $\Delta'_2(x) \sqsubseteq \ell$, and $\Delta'_1(x) \neq \Delta'_2(x)$. We have already shown above that from $\Delta'_1(x) \sqsubseteq \ell$ and (2), it follows $\Delta'_2(x) = \Delta'_1(x)$, which contradicts $\Delta'_1(x) \neq \Delta'_2(x)$.
- $\Delta'_1(x) \sqsubseteq \ell$, $\Delta'_2(x) \sqsubseteq \ell$, and $m'_1(x) \neq m'_2(x)$. From the rule, it follows that $\Delta'_1 = \Delta_1[x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e)]$ and $\Delta'_2 = \Delta_2[x \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \Delta_2(e)]$. From this and $\Delta'_1(x) \sqsubseteq \ell$, it follows that $\bigwedge_{y \in \mathit{free}(e)} \Delta_1(y) \sqsubseteq \ell$. From this and (2), it follows that $\bigwedge_{y \in \mathit{free}(e)} \Delta_2(y) \sqsubseteq \ell$. From $\bigwedge_{y \in \mathit{free}(e)} \Delta_1(y) \sqsubseteq \ell$, $\bigwedge_{y \in \mathit{free}(e)} \Delta_2(y) \sqsubseteq \ell$, and (2), it follows that $\bigwedge_{y \in \mathit{free}(e)} m_1(y) = m_2(y)$. From this, it follows that $[\![e]\!](m_1) = [\![e]\!](m_2)$. From this, $m'_1 = m_1[x \mapsto [\![e]\!](m_1)]$, and $m'_2 = m_2[x \mapsto [\![e]\!](m_2)]$, it follows that $m'_1(x) = m'_2(x)$, leading to a contradiction.

- Rule F-IfTrue. From the rule, it follows that $c_1 = \textbf{if } e \textbf{ then } c' \textbf{ else } c''$. From this and (4), it follows that $c_2 = \textbf{if } e \textbf{ then } c' \textbf{ else } c''$. From $\langle \Delta_1, c_1, m_1, \langle s_1, \mathit{ctx}_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta'_1, c'_1, m'_1, \langle s'_1, \mathit{ctx}'_1\rangle\rangle$ and $c_1 = \textbf{if } e \textbf{ then } c' \textbf{ else } c''$, it follows that $\Delta'_1 = \Delta_1[\mathtt{pc}_u \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e)]$, $m_1 = m'_1$, $\tau_1 = \epsilon$, $c'_1 = c'$; **set pc to** $\Delta_1(\mathtt{pc}_u)$, and $\langle s_1, \mathit{ctx}_1\rangle = \langle s'_1, \mathit{ctx}'_1\rangle$. From the rule, it also follows that $[\![e]\!](m_1) = \mathbf{tt}$. There are two cases:
  - $[\![e]\!](m_2) = \mathbf{tt}$. From this, $\langle \Delta_2, c_2, m_2, \langle s_2, \mathit{ctx}_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta'_2, c'_2, m'_2, \langle s'_2, \mathit{ctx}'_2\rangle\rangle$ and $c_2 = \textbf{if } e \textbf{ then } c' \textbf{ else } c''$, it follows that $\Delta'_2 = \Delta_2[\mathtt{pc}_u \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \Delta_2(e)]$, $m_2 = m'_2$, $\tau_2 = \epsilon$, $c'_2 = c'$; **set pc to** $\Delta_2(\mathtt{pc}_u)$, and $\langle s_2, \mathit{ctx}_2\rangle = \langle s'_2, \mathit{ctx}'_2\rangle$. Assume, for contradiction's sake, that $\langle \Delta'_1, c'_1, m'_1, \langle s'_1, \mathit{ctx}'_1\rangle\rangle \not\approx_\ell \langle \Delta'_2, c'_2, m'_2, \langle s'_2, \mathit{ctx}'_2\rangle\rangle$. From this, $m_1 = m'_1$, $m_2 = m'_2$, $\langle s_1, \mathit{ctx}_1\rangle = \langle s'_1, \mathit{ctx}'_1\rangle$, $\langle s_2, \mathit{ctx}_2\rangle = \langle s'_2, \mathit{ctx}'_2\rangle$, and (2), there are two cases:
    * $\Delta'_1(\mathtt{pc}_u) \sqsubseteq \ell$ and $\Delta'_2(\mathtt{pc}_u) \not\sqsubseteq \ell$ (or vice versa). From the rule, it follows that $\Delta'_1 = \Delta_1[\mathtt{pc}_u \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e)]$ and $\Delta'_2 = \Delta_2[\mathtt{pc}_u \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \Delta_2(e)]$. From this and $\Delta'_1(\mathtt{pc}_u) \sqsubseteq \ell$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e) \sqsubseteq \ell$. From this, $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and $\bigwedge_{y \in \mathit{free}(e)} \Delta_1(y) \sqsubseteq \ell$. From $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and (2), it follows that $\Delta_1(\mathtt{pc}_u) = \Delta_2(\mathtt{pc}_u)$. From $\bigwedge_{y \in \mathit{free}(e)} \Delta_1(y) \sqsubseteq \ell$ and (2), it follows that $\bigwedge_{y \in \mathit{free}(e)} \Delta_1(y) = \Delta_2(y)$. From $\Delta_1(\mathtt{pc}_u) = \Delta_2(\mathtt{pc}_u)$ and $\bigwedge_{y \in \mathit{free}(e)} \Delta_1(y) = \Delta_2(y)$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e) = \Delta_2(\mathtt{pc}_u) \sqcup \Delta_2(e)$. From this, $\Delta'_1 = \Delta_1[\mathtt{pc}_u \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e)]$, and $\Delta'_2 = \Delta_2[\mathtt{pc}_u \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \Delta_2(e)]$, it follows that $\Delta'_2(x) = \Delta'_1(x)$. From this and $\Delta'_1(\mathtt{pc}_u) \sqsubseteq \ell$, it follows that $\Delta'_2(\mathtt{pc}_u) \sqsubseteq \ell$, leading to a contradiction.
    * $\Delta'_1(\mathtt{pc}_u) \sqsubseteq \ell$, $\Delta'_2(\mathtt{pc}_u) \sqsubseteq \ell$, and $\Delta'_1(\mathtt{pc}_u) \neq \Delta'_2(\mathtt{pc}_u)$. We have already shown above that from $\Delta'_1(\mathtt{pc}_u) \sqsubseteq \ell$ and (2), it follows $\Delta'_2(\mathtt{pc}_u) = \Delta'_1(\mathtt{pc}_u)$, which contradicts $\Delta'_1(\mathtt{pc}_u) \neq \Delta'_2(\mathtt{pc}_u)$.
  - $[\![e]\!](m_2) = \mathbf{ff}$. From this, $\langle \Delta_2, c_2, m_2, \langle s_2, \mathit{ctx}_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta'_2, c'_2, m'_2, \langle s'_2, \mathit{ctx}'_2\rangle\rangle$ and $c_2 = \textbf{if } e \textbf{ then } c' \textbf{ else } c''$, it follows that $\Delta'_2 = \Delta_2[\mathtt{pc}_u \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \Delta_2(e)]$, $m_2 = m'_2$, $\tau_2 = \epsilon$, $c'_2 = c''$; **set pc to** $\Delta_2(\mathtt{pc}_u)$, and $\langle s_2, \mathit{ctx}_2\rangle = \langle s'_2, \mathit{ctx}'_2\rangle$. The rest of the proof is similar to the case above.
- Rule F-IfFalse. The proof of this case is similar to that of F-IfTrue.
- Rule F-WhileTrue. The proof of this case is similar to that of F-IfTrue.
- Rule F-WhileFalse. The proof of this case is similar to that of F-IfTrue.
- Rule F-Select. From the rule, it follows that $c_1 = \|x \leftarrow \mathtt{SELECT}\ \varphi\|$. From this and (4), it follows that $c_2 = \|x \leftarrow \mathtt{SELECT}\ \varphi\|$. From $\langle \Delta_1, c_1, m_1, \langle s_1, \mathit{ctx}_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta'_1, c'_1, m'_1, \langle s'_1, \mathit{ctx}'_1\rangle\rangle$ and $c_1 = \|x \leftarrow q\|$, it follows that $\Delta'_1 = \Delta_1[x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup L_\mathcal{Q}(\Delta_1, q_1) \sqcup \bigsqcup_{v \in \mathit{free}(\varphi)} \Delta_1(v)]$, $m'_1 = m_1[x \mapsto r_1]$, $\tau_1 = \epsilon$, $c'_1 = \varepsilon$, where $\mathit{free}(\varphi) = \{v_1, \ldots, v_n\}$, $q_1 = \mathtt{SELECT}\ \varphi[v_1 \mapsto [\![v_1]\!](m_1), \ldots, v_n \mapsto [\![v_n]\!](m_1),]$, and $[\![q_1]\!](\langle s_1, \mathit{ctx}_1\rangle) = \langle\langle s'_1, \mathit{ctx}'_1\rangle, r_1, \epsilon\rangle$. From $\langle \Delta_2, c_2, m_2, \langle s_2, \mathit{ctx}_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta'_2, c'_2, m'_2, \langle s'_2, \mathit{ctx}'_2\rangle\rangle$ and $c_2 = \|x \leftarrow q\|$, it follows that $\Delta'_2 = \Delta_2[x \mapsto \Delta_2(\mathtt{pc}_u) \sqcup L_\mathcal{Q}(\Delta_2, q_2) \sqcup \bigsqcup_{v \in \mathit{free}(\varphi)} \Delta_2(v)]$, $m'_2 = m_2[x \mapsto r_2]$, $\tau_2 = \epsilon$, $c'_2 = \varepsilon$, where $\mathit{free}(\varphi) = \{v_1, \ldots, v_n\}$, $q_2 = \mathtt{SELECT}\ \varphi[v_1 \mapsto [\![v_1]\!](m_2), \ldots, v_n \mapsto [\![v_n]\!](m_2),]$, and $[\![q_2]\!](\langle s_2, \mathit{ctx}_2\rangle) = \langle\langle s'_2, \mathit{ctx}'_2\rangle, r_2, \epsilon\rangle$. Assume, for contradiction's sake, that $\langle \Delta'_1, c'_1, m'_1, \langle s'_1, \mathit{ctx}'_1\rangle\rangle \not\approx_\ell \langle \Delta'_2, c'_2, m'_2, \langle s'_2, \mathit{ctx}'_2\rangle\rangle$. From this and (2), there are three cases:
  - $\Delta'_1(x) \sqsubseteq \ell$ and $\Delta'_2(x) \not\sqsubseteq \ell$ (or vice versa). From $\Delta'_1(x) \sqsubseteq \ell$ and $\Delta'_1 = \Delta_1[x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup L_\mathcal{Q}(\Delta_1, q_1) \sqcup \bigsqcup_{v \in \mathit{free}(\varphi)} \Delta_1(v)]$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup L_\mathcal{Q}(\Delta_1, q_1) \sqcup \bigsqcup_{v \in \mathit{free}(\varphi)} \Delta_1(v) \sqsubseteq \ell$. From this, it follows that $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$, $L_\mathcal{Q}(\Delta_1, q_1) \sqsubseteq \ell$, and $\bigwedge_{v \in \mathit{free}(\varphi)} \Delta_1(v) \sqsubseteq \ell$. From $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and (2), it follows that $\Delta_2(\mathtt{pc}_u) \sqsubseteq \ell$. From $\bigwedge_{v \in \mathit{free}(\varphi)} \Delta_1(v) \sqsubseteq \ell$ and (2), it follows that $\bigwedge_{v \in \mathit{free}(\varphi)} \Delta_2(v) \sqsubseteq \ell$ and $\bigwedge_{v \in \mathit{free}(\varphi)} m_1(v) = m_2(v)$. From $\bigwedge_{v \in \mathit{free}(\varphi)} m_1(v) = m_2(v)$, it follows that $q_1 = q_2$. From $q_1 = q_2$, $L_\mathcal{Q}(\Delta_1, q_1) \sqsubseteq \ell$, (2), and Proposition H.2, it follows that $L_\mathcal{Q}(\Delta_2, q_2) \sqsubseteq \ell$. From $\Delta_2(\mathtt{pc}_u) \sqsubseteq \ell$, $L_\mathcal{Q}(\Delta_2, q_2) \sqsubseteq \ell$, $\bigwedge_{v \in \mathit{free}(\varphi)} \Delta_2(v) \sqsubseteq \ell$, and point (3) in the notion of disclosure order, it follows that $\Delta_2(\mathtt{pc}_u) \sqcup L_\mathcal{Q}(\Delta_2, q_2) \sqcup \bigsqcup_{v \in \mathit{free}(\varphi)} \Delta_2(v) \sqsubseteq \ell$. From this and $\Delta'_2 = \Delta_2[x \mapsto \Delta_2(\mathtt{pc}_u) \sqcup L_\mathcal{Q}(\Delta_2, q_2) \sqcup \bigsqcup_{v \in \mathit{free}(\varphi)} \Delta_2(v)]$, it follows that $\Delta'_2(x) \sqsubseteq \ell$, leading to a contradiction.
  - $\Delta'_1(x) \sqsubseteq \ell$, $\Delta'_1(x) \sqsubseteq \ell$, and $\Delta'_1(x) \neq \Delta'_2(x)$. From $\Delta'_1(x) \sqsubseteq \ell$ and $\Delta'_1 = \Delta_1[x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup L_\mathcal{Q}(\Delta_1, q_1) \sqcup \bigsqcup_{v \in \mathit{free}(\varphi)} \Delta_1(v)]$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup L_\mathcal{Q}(\Delta_1, q_1) \sqcup \bigsqcup_{v \in \mathit{free}(\varphi)} \Delta_1(v) \sqsubseteq \ell$. From this, it follows that $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$, $L_\mathcal{Q}(\Delta_1, q_1) \sqsubseteq \ell$, and $\bigwedge_{v \in \mathit{free}(\varphi)} \Delta_1(v) \sqsubseteq \ell$. From $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and (2), it follows that $\Delta_2(\mathtt{pc}_u) = \Delta_1(\mathtt{pc}_u)$. From $\bigwedge_{v \in \mathit{free}(\varphi)} \Delta_1(v) \sqsubseteq \ell$ and (2), it follows that $\bigwedge_{v \in \mathit{free}(\varphi)} \Delta_2(v) = \Delta_1(v)$ and $\bigwedge_{v \in \mathit{free}(\varphi)} m_1(v) = m_2(v)$. From $\bigwedge_{v \in \mathit{free}(\varphi)} m_1(v) = m_2(v)$, it follows that $q_1 = q_2$. From $L_\mathcal{Q}(\Delta_1, q_1) \sqsubseteq \ell$, it follows that $\bigsqcup_{Q \in \mathit{supp}_{D,\Gamma}(q_1)} \bigsqcup_{q' \in Q} \Delta_1(q') \sqsubseteq \ell$. From this, $\bigwedge_{Q \in \mathit{supp}_{D,\Gamma}(q_1)} \bigwedge_{q' \in Q} \Delta_1(q') \sqsubseteq \ell$. From this and (2), it follows that $\bigwedge_{Q \in \mathit{supp}_{D,\Gamma}(q_1)} \bigwedge_{q' \in Q} \Delta_1(q') = \Delta_2(q')$. From this, it follows that $\bigwedge_{Q \in \mathit{supp}_{D,\Gamma}(q_1)} \bigwedge_{q' \in Q} \Delta_1(q') = \bigwedge_{Q \in \mathit{supp}_{D,\Gamma}(q_1)} \bigwedge_{q' \in Q} \Delta_2(q')$. From this and $q_1 = q_2$, it follows that $\bigwedge_{Q \in \mathit{supp}_{D,\Gamma}(q_1)} \bigwedge_{q' \in Q} \Delta_1(q') = \bigwedge_{Q \in \mathit{supp}_{D,\Gamma}(q_2)} \bigwedge_{q' \in Q} \Delta_2(q')$. From this, it follows that $L_\mathcal{Q}(\Delta_1, q_1) = L_\mathcal{Q}(\Delta_2, q_2)$. From $\Delta_2(\mathtt{pc}_u) = \Delta_1(\mathtt{pc}_u)$, $\bigwedge_{v \in \mathit{free}(\varphi)} \Delta_2(v) = \Delta_1(v)$, $L_\mathcal{Q}(\Delta_1, q_1) = L_\mathcal{Q}(\Delta_2, q_2)$, $\Delta'_1 = \Delta_1[x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup L_\mathcal{Q}(\Delta_1, q_1) \sqcup \bigsqcup_{v \in \mathit{free}(\varphi)} \Delta_1(v)]$, and $\Delta'_2 = \Delta_2[x \mapsto \Delta_2(\mathtt{pc}_u) \sqcup L_\mathcal{Q}(\Delta_2, q_2) \sqcup \bigsqcup_{v \in \mathit{free}(\varphi)} \Delta_2(v)]$, it follows that $\Delta'_1(x) = \Delta'_2(x)$, leading to a contradiction.
  - $\Delta'_1(x) \sqsubseteq \ell$, $\Delta'_1(x) \sqsubseteq \ell$, and $m'_1(x) \neq m'_2(x)$. From $\Delta'_1(x) \sqsubseteq \ell$ and $\Delta'_1 = \Delta_1[x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup L_\mathcal{Q}(\Delta_1,$

$q_1) \sqcup \bigsqcup_{v \in \mathit{free}(\varphi)} \Delta_1(v)]$, it follows that $\Delta_1(\mathsf{pc}_u) \sqcup L_{\mathcal{Q}}(\Delta_1, q_1) \sqcup \bigsqcup_{v \in \mathit{free}(\varphi)} \Delta_1(v) \sqsubseteq \ell$. From this and (2), it follows that $\bigwedge_{v \in \mathit{free}(\varphi)} m_1(v) = m_2(v)$. From this, it follows that $q_1 = q_2$. From $\Delta_1(\mathsf{pc}_u) \sqcup L_{\mathcal{Q}}(\Delta_1, q_1) \sqcup \bigsqcup_{v \in \mathit{free}(\varphi)} \Delta_1(v) \sqsubseteq \ell$, it also follows that $L_{\mathcal{Q}}(\Delta_1, q_1) \sqsubseteq \ell$. From this, $q_1 = q_2$, (2), and Proposition H.2, it follows that $L_{\mathcal{Q}}(\Delta_2, q_2) \sqsubseteq \ell$. From this, $L_{\mathcal{Q}}(\Delta_2, q_2) \sqsubseteq \ell$, and (2), it follows that $[q_1]^{db_1} = [q_2]^{db_2}$, where db_1 and db_2 are the database states in s_1 and s_2 respectively. From the database semantics, it follows that $m_1'(x) = r_1 = [q_1]^{db_1}$ and $m_2'(x) = r_2 = [q_2]^{db_2}$. From this and $[db_1]^{s_1} = [db_2]^{s_2}$, it follows that $m_1'(x) = m_2'(x)$, leading to a contradiction.

- Rule F-UPDATEDATABASEOK. Without loss of generality, we assume that the query is an `INSERT` query. From the rule, it follows that $c_1 = \|x \leftarrow T \oplus \overline{e}\|$. From this and (4), it follows that $c_2 = \|x \leftarrow T \oplus \overline{e}\|$. From $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1 \rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1' \rangle\rangle$ and $c_1 = \|x \leftarrow T \oplus \overline{e}\|$, it follows that $\Delta_1' = \Delta_1[T(\overline{v}_1) \mapsto \Delta_1(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_1(e_i), x \mapsto \Delta_1(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_1(e_i)]$, $m_1' = m_1[x \mapsto r_1]$, $\tau_1 = \epsilon$, $c_1' = \varepsilon$, where $\overline{e} = (e_1, \ldots, e_n)$, $\overline{v}_1 = (\llbracket e_1 \rrbracket(m_1), \ldots, \llbracket e_n \rrbracket(m_1))$, $q_1 = T \oplus \overline{v}_1$, and $\llbracket q_1 \rrbracket(\langle s_1, ctx_1 \rangle) = \langle \langle s_1', ctx_1' \rangle, r_1, \epsilon \rangle$. From $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2 \rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2' \rangle\rangle$ and $c_2 = \|x \leftarrow T \oplus \overline{e}\|$, it follows that $\Delta_2' = \Delta_2[T(\overline{v}_2) \mapsto \Delta_2(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_2(e_i), x \mapsto \Delta_2(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_2(e_i)]$, $m_2' = m_2[x \mapsto r_2]$, $\tau_2 = \epsilon$, $c_2' = \varepsilon$, where $\overline{e} = (e_1, \ldots, e_n)$, $\overline{v}_2 = (\llbracket e_1 \rrbracket(m_2), \ldots, \llbracket e_n \rrbracket(m_2))$, $q_2 = T \oplus \overline{v}_2$, and $\llbracket q_2 \rrbracket(\langle s_2, ctx_2 \rangle) = \langle \langle s_2', ctx_2' \rangle, r_2, \epsilon \rangle$. Assume, for contradiction's sake, that $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1' \rangle\rangle \not\approx_\ell \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2' \rangle\rangle$. From this and (2), there are several cases:
  1) $\Delta_1'(x) \sqsubseteq \ell$ and $\Delta_2'(x) \not\sqsubseteq \ell$ (or vice versa). From $\Delta_1'(x) \sqsubseteq \ell$ and $\Delta_1'(x) = \Delta_1(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_1(e_i)$, it follows that $\Delta_1(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_1(e_i) \sqsubseteq \ell$. From this, it follows that $\Delta_1(\mathsf{pc}_u) \sqsubseteq \ell$ and $\bigwedge_{1 \le i \le n} \Delta_1(e_i) \sqsubseteq \ell$. From $\Delta_1(\mathsf{pc}_u) \sqsubseteq \ell$ and (2), it follows that $\Delta_2(\mathsf{pc}_u) \sqsubseteq \ell$. From $\bigwedge_{1 \le i \le n} \Delta_1(e_i) \sqsubseteq \ell$ and (2), it follows that $\bigwedge_{1 \le i \le n} \Delta_2(e_i) \sqsubseteq \ell$. From this, $\Delta_2(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_2(e_i) \sqsubseteq \ell$. From this and $\Delta_2'(x) = \Delta_2(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_2(e_i)$, it follows that $\Delta_2'(x) \sqsubseteq \ell$, leading to a contradiction.
  2) $\Delta_1'(x) \sqsubseteq \ell$, $\Delta_2'(x) \sqsubseteq \ell$, and $\Delta_1(x) \neq \Delta_2(x)$. From $\Delta_1'(x) \sqsubseteq \ell$ and $\Delta_1'(x) = \Delta_1(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_1(e_i)$, it follows that $\Delta_1(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_1(e_i) \sqsubseteq \ell$. From this, it follows that $\Delta_1(\mathsf{pc}_u) \sqsubseteq \ell$ and $\bigwedge_{1 \le i \le n} \Delta_1(e_i) \sqsubseteq \ell$. From $\Delta_1(\mathsf{pc}_u) \sqsubseteq \ell$ and (2), it follows that $\Delta_2(\mathsf{pc}_u) = \Delta_1(\mathsf{pc}_u)$. From $\bigwedge_{1 \le i \le n} \Delta_1(e_i) \sqsubseteq \ell$ and (2), it follows that $\bigwedge_{1 \le i \le n} \Delta_2(e_i) = \Delta_1(e_i)$. From this, $\Delta_2(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_2(e_i) = \Delta_1(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_1(e_i)$. From this and $\Delta_2'(x) = \Delta_2(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_2(e_i)$, it follows that $\Delta_2'(x) = \Delta_1'(x)$, leading to a contradiction.
  3) $\Delta_1'(x) \sqsubseteq \ell$, $\Delta_2'(x) \sqsubseteq \ell$, and $m_1'(x) \neq m_2'(x)$. From the database semantics, both r_1 and r_2 are $\top$. From this, $m_1'(x) = r_1$, and $m_2'(x) = r_2$, it follows that $m_1'(x) = m_2'(x)$, leading to a contradiction.
  4) $\Delta_1'(T(\overline{v}_1)) \sqsubseteq \ell$ and $\Delta_2'(T(\overline{v}_1)) \not\sqsubseteq \ell$. From $\Delta_1'(T(\overline{v}_1)) \sqsubseteq \ell$ and $\Delta_1'(T(\overline{v}_1)) = \Delta_1(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_1(e_i)$, it follows that $\Delta_1(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_1(e_i) \sqsubseteq \ell$. From this, $\Delta_1(\mathsf{pc}_u) \sqsubseteq \ell$ and $\bigwedge_{1 \le i \le n} \Delta_1(e_i) \sqsubseteq \ell$. From this and (2), it follows that $\Delta_2(\mathsf{pc}_u) \sqsubseteq \ell$, $\bigwedge_{1 \le i \le n} \Delta_2(e_i) \sqsubseteq \ell$ and $\bigwedge_{1 \le i \le n} \llbracket e_i \rrbracket(m_1) = \llbracket e_i \rrbracket(m_2)$. From $\bigwedge_{1 \le i \le n} \llbracket e_i \rrbracket(m_1) = \llbracket e_i \rrbracket(m_2)$, it follows that $\overline{v}_1 = \overline{v}_2$. From $\Delta_2(\mathsf{pc}_u) \sqsubseteq \ell$ and $\bigwedge_{1 \le i \le n} \Delta_2(e_i) \sqsubseteq \ell$, it follows that $\Delta_2(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_2(e_i) \sqsubseteq \ell$. From this, $\Delta_2'(T(\overline{v}_2)) = \Delta_2(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_2(e_i)$, and $\overline{v}_1 = \overline{v}_2$, it follows that $\Delta_2'(T(\overline{v}_2)) \sqsubseteq \ell$, leading to a contradiction.
  5) $\Delta_1'(T(\overline{v}_2)) \sqsubseteq \ell$ and $\Delta_2'(T(\overline{v}_2)) \not\sqsubseteq \ell$. There are two cases:
     - $\overline{v}_1 = \overline{v}_2$. We already proved above (case 4) that this leads to a contradiction.
     - $\overline{v}_1 \neq \overline{v}_2$. From $\Delta_2'(T(\overline{v}_2)) \not\sqsubseteq \ell$ and $\Delta_2'(T(\overline{v}_2)) = \Delta_2(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_2(e_i)$, it follows that $\Delta_2(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_2(e_i) \not\sqsubseteq \ell$. From (3), it follows that $\Delta_2(\mathsf{pc}_u) \sqsubseteq \ell$. From this and $\Delta_2(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_2(e_i) \not\sqsubseteq \ell$, it follows that $\bigsqcup_{1 \le i \le n} \Delta_2(e_i) \not\sqsubseteq \ell$. From the rule, it follows that $\bigsqcup_{1 \le i \le n} \Delta_2(e_i) \sqsubseteq \Delta_2(T(\overline{v}_2))$. From this and $\bigsqcup_{1 \le i \le n} \Delta_2(e_i) \not\sqsubseteq \ell$, it follows that $\Delta_2(T(\overline{v}_2)) \not\sqsubseteq \ell$. From this and (2), it follows that $\Delta_1(T(\overline{v}_2)) \not\sqsubseteq \ell$. From $\Delta_1' = \Delta_1[T(\overline{v}_1) \mapsto \Delta_1(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_1(e_i), x \mapsto \Delta_1(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_1(e_i)]$ and $\overline{v}_1 \neq \overline{v}_2$, it follows that $\Delta_1'(T(\overline{v}_2)) = \Delta_1(T(\overline{v}_2))$. From this and $\Delta_1'(T(\overline{v}_2)) \sqsubseteq \ell$, it follows that $\Delta_1(T(\overline{v}_2)) \sqsubseteq \ell$. This contradicts $\Delta_1(T(\overline{v}_2)) \not\sqsubseteq \ell$.
  6) $\Delta_2'(T(\overline{v}_2)) \sqsubseteq \ell$ and $\Delta_1'(T(\overline{v}_2)) \not\sqsubseteq \ell$. The proof of this case is similar to that of case 4.
  7) $\Delta_2'(T(\overline{v}_1)) \sqsubseteq \ell$ and $\Delta_1'(T(\overline{v}_1)) \not\sqsubseteq \ell$. The proof of this case is similar to that of case 5.
  8) $\Delta_1'(T(\overline{v}_1)) \sqsubseteq \ell$, $\Delta_2'(T(\overline{v}_1)) \sqsubseteq \ell$, and $\Delta_1'(T(\overline{v}_1)) \neq \Delta_2'(T(\overline{v}_1))$. From $\Delta_1'(T(\overline{v}_1)) \sqsubseteq \ell$ and $\Delta_1'(T(\overline{v}_1)) = \Delta_1(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_1(e_i)$, it follows that $\Delta_1(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_1(e_i) \sqsubseteq \ell$. From this, $\Delta_1(\mathsf{pc}_u) \sqsubseteq \ell$ and $\bigwedge_{1 \le i \le n} \Delta_1(e_i) \sqsubseteq \ell$. From this and (2), it follows that $\Delta_2(\mathsf{pc}_u) = \Delta_1(\mathsf{pc}_u)$, $\bigwedge_{1 \le i \le n} \Delta_2(e_i) = \Delta_1(e_i)$ and $\bigwedge_{1 \le i \le n} \llbracket e_i \rrbracket(m_1) = \llbracket e_i \rrbracket(m_2)$. From $\bigwedge_{1 \le i \le n} \llbracket e_i \rrbracket(m_1) = \llbracket e_i \rrbracket(m_2)$, it follows that $\overline{v}_1 = \overline{v}_2$. From $\Delta_2(\mathsf{pc}_u) = \Delta_1(\mathsf{pc}_u)$ and $\bigwedge_{1 \le i \le n} \Delta_2(e_i) = \Delta_1(e_i)$, it follows that $\Delta_1(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_1(e_i) = \Delta_2(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_2(e_i)$. From this, $\overline{v}_1 = \overline{v}_2$, $\Delta_1'(T(\overline{v}_1)) = \Delta_1(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_1(e_i)$, $\Delta_2'(T(\overline{v}_2)) = \Delta_2(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_2(e_i)$, it follows that $\Delta_2'(T(\overline{v}_1)) = \Delta_1(T(\overline{v}_1))$.
  9) $\Delta_1'(T(\overline{v}_2)) \sqsubseteq \ell$, $\Delta_2'(T(\overline{v}_2)) \sqsubseteq \ell$, and $\Delta_1'(T(\overline{v}_2)) \neq \Delta_2'(T(\overline{v}_2))$. The proof is similar to that of case 8.
  10) There is a query q such that $L_{\mathcal{Q}}(\Delta_1', q) \sqsubseteq \ell$, $L_{\mathcal{Q}}(\Delta_2', q) \sqsubseteq \ell$, and $[q]^{db_1'} \neq [q]^{db_2'}$, where db_1' and db_2' are the databases in s_1' and s_2' respectively. From $L_{\mathcal{Q}}$'s definition, it follows $\bigsqcup_{Q \in \mathit{supp}_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} \Delta_1'(q') \sqsubseteq \ell$ and $\bigsqcup_{Q \in \mathit{supp}_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} \Delta_2'(q') \sqsubseteq \ell$. There are two cases:
      - There is a $Q \in \mathit{supp}_{D,\Gamma}(q)$ such that $T(\overline{v}_1) \notin Q$ and $T(\overline{v}_2) \notin Q$. From $L_{\mathcal{Q}}(\Delta_1', q) \sqsubseteq \ell$, it follows that $\bigwedge_{q' \in Q} \Delta_1'(q') \sqsubseteq \ell$. From $T(\overline{v}_1) \notin Q$ and $\Delta_1' = \Delta_1[T(\overline{v}_1) \mapsto \Delta_1(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_1(e_i), x \mapsto \Delta_1(\mathsf{pc}_u) \sqcup \bigsqcup_{1 \le i \le n} \Delta_1(e_i)]$, it follows that $\bigwedge_{q' \in Q} \Delta_1'(q') = \Delta_1(q')$. From this and $\bigwedge_{q' \in Q} \Delta_1'(q') \sqsubseteq \ell$, it follows that $\bigwedge_{q' \in Q} \Delta_1(q') \sqsubseteq \ell$. From this, $Q \subset RC^{pred}$, and $L_{\mathcal{Q}}(\Delta, q'') = \Delta(q'')$ for any $q'' \in RC^{pred}$ (Proposition H.3),

it follows that $\bigwedge_{q' \in Q} L_{\mathcal{Q}}(\Delta_1, q') \sqsubseteq \ell$. From this and (2), it follows that $\bigwedge_{q' \in Q} [q']^{db'_1} = [q']^{db'_2}$, where db'_1 and db'_2 are the database in s'_1 and s'_2 respectively. From this, $T(\overline{v}_1) \notin Q$, $T(\overline{v}_2) \notin Q$, and the fact that we modify only the values of $T(\overline{v}_1)$ and $T(\overline{v}_2)$, it follows that $\bigwedge_{q' \in Q} [q']^{s'_1} = [q']^{s'_2}$. From this and Q determines q, it follows that $[q]^{s_1} = [q]^{s_2}$, leading to a contradiction.

  - For all $Q \in \mathit{supp}_{D,\Gamma}(q)$, $T(\overline{v}_1) \in Q$ or $T(\overline{v}_2) \in Q$. Assume that $T(\overline{v}_1) \in Q$ (the proof in case $T(\overline{v}_2) \in Q$ is analogous). From $\bigsqcup_{Q \in \mathit{supp}_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} \Delta'_1(q') \sqsubseteq \ell$, it follows that $\bigwedge_{q' \in Q} \Delta'_1(q') \sqsubseteq \ell$. From this and $T(\overline{v}_1) \in Q$, it follows that $\Delta'_1(T(\overline{v}_1)) \sqsubseteq \ell$. From this and $\Delta'_1(T(\overline{v}_1)) = \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{1 \leq i \leq n} \Delta_1(e_i)$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{1 \leq i \leq n} \Delta_1(e_i) \sqsubseteq \ell$. From this, it follows that $\bigwedge_{1 \leq i \leq n} \Delta_1(e_i) \sqsubseteq \ell$. From this and (2), it follows that $\bigwedge_{1 \leq i \leq n} [\![e_i]\!](m_1) = [\![e_i]\!](m_2)$. From this, it follows that $\overline{v}_1 = \overline{v}_2$. From $\bigsqcup_{Q \in \mathit{supp}_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} \Delta'_1(q') \sqsubseteq \ell$ and $Q \in \mathit{supp}_{D,\Gamma}(q)$, it follows that $\bigwedge_{q' \in Q} \Delta'_1(q') \sqsubseteq \ell$. Let q'' be a query in $Q \setminus \{T(\overline{v}_1)\}$. From this and $\bigwedge_{q' \in Q} \Delta'_1(q') \sqsubseteq \ell$, it follows that $\Delta'_1(q'') \sqsubseteq \ell$. From this, $\overline{v}_1 = \overline{v}_2$, $q'' \in Q \setminus \{T(\overline{v}_1)\}$, and $\Delta'_1 = \Delta_1[T(\overline{v}_1) \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{1 \leq i \leq n} \Delta_1(e_i), x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{1 \leq i \leq n} \Delta_1(e_i)]$, it follows that $\Delta'_1(q'') = \Delta_1(q'')$. From this and $\Delta'_1(q'') \sqsubseteq \ell$, it follows that $\Delta_1(q'') \sqsubseteq \ell$. From this, $q'' \in \mathit{RC}^{\mathit{pred}}$, and $L_{\mathcal{Q}}(\Delta, q) = \Delta(q)$ for all $q \in \mathit{RC}^{\mathit{pred}}$ (Proposition H.3), it follows that $L_{\mathcal{Q}}(\Delta_1, q'') \sqsubseteq \ell$. From this and (2), it follows that $[q'']^{db_1} = [q'']^{db_2}$, where db_1 and db_2 are the databases in s_1 and s_2 respectively. From this, $\overline{v}_1 = \overline{v}_2$, and the fact that the update operation only modifies the value of $T(\overline{v}_1)$ and $T(\overline{v}_2)$, it follows that $[q'']^{db'_1} = [q'']^{db'_2}$, where db'_1 and db'_2 are the databases in s'_1 and s'_2 respectively. Since q'' is an arbitrary query in $Q \setminus \{T(\overline{v}_1)\}$, it follows that $[q'']^{db'_1} = [q'']^{db'_2}$ for all $q'' \in Q \setminus \{T(\overline{v}_1)\}$. From $\overline{v}_1 = \overline{v}_2$ and the database semantics, it also follows that $[T(\overline{v}_1)]^{db'_1} = [T(\overline{v}_1)]^{db'_2} = \top$. From this, $[q'']^{db'_1} = [q'']^{db'_2}$ for all $q'' \in Q \setminus \{T(\overline{v}_1)\}$, and $Q = \{T(\overline{v}_1)\} \cup (Q \setminus \{T(\overline{v}_1)\})$, it follows that $[q'']^{db'_1} = [q'']^{db'_2}$ for all $q'' \in Q$. From this and $Q \in \mathit{supp}_{D,\Gamma}(q)$ (and therefore Q determines q), it follows that $[q]^{db'_1} = [q]^{db'_2}$, leading to a contradiction.

- Rule F-UPDATECONFIGURATIONOK. From the rule, it follows that $c_1 = \|x \leftarrow q\|$. From this and (4), it follows that $c_2 = \|x \leftarrow q\|$. From $\langle \Delta_1, c_1, m_1, \langle s_1, \mathit{ctx}_1 \rangle \rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta'_1, c'_1, m'_1, \langle s'_1, \mathit{ctx}'_1 \rangle \rangle$ and $c_1 = \|x \leftarrow q\|$, it follows that $\Delta'_1 = \Delta_1[x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in \mathit{vars}(q)} \Delta_1(v)]$, $m'_1 = m_1[x \mapsto r_1]$, $q'_1 = q[v_1 \mapsto [\![v_1]\!](m_1), \ldots, v_n \mapsto [\![v_n]\!](m_1)]$ $\tau_1 = \langle \mathit{public}, q'_1 \rangle$, $c'_1 = \varepsilon$, where $[\![q'_1]\!](\langle s_1, \mathit{ctx}_1 \rangle) = \langle \langle s'_1, \mathit{ctx}'_1 \rangle, r_1, \epsilon \rangle$. From $\langle \Delta_2, c_2, m_2, \langle s_2, \mathit{ctx}_2 \rangle \rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta'_2, c'_2, m'_2, \langle s'_2, \mathit{ctx}'_2 \rangle \rangle$ and $c_2 = \|x \leftarrow q\|$, it follows that $\Delta'_2 = \Delta_2[x \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in \mathit{vars}(q)} \Delta_2(v)]$, $m'_2 = m_2[x \mapsto r_2]$, $q'_2 = q[v_1 \mapsto [\![v_1]\!](m_2), \ldots, v_n \mapsto [\![v_n]\!](m_2)]$ $\tau_2 = \langle \mathit{public}, q'_2 \rangle$, $c'_2 = \varepsilon$, where $[\![q'_2]\!](\langle s_2, \mathit{ctx}_2 \rangle) = \langle \langle s'_2, \mathit{ctx}'_2 \rangle, r_2, \epsilon \rangle$. Note that the execution of the query q alters only the database configuration; it does not modify the content of the database. Assume, for contradiction's sake, that $\langle \Delta'_1, c'_1, m'_1, \langle s'_1, \mathit{ctx}'_1 \rangle \rangle \not\approx_\ell \langle \Delta'_2, c'_2, m'_2, \langle s'_2, \mathit{ctx}'_2 \rangle \rangle$. There are three cases:
  - $\Delta'_1(x) \sqsubseteq \ell$ and $\Delta'_2(x) \not\sqsubseteq \ell$ (or vice versa). From $\Delta'_1(x) \sqsubseteq \ell$ and $\Delta'_1 = \Delta_1[x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in \mathit{vars}(q)} \Delta_1(v)]$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in \mathit{vars}(q)} \Delta_1(v) \sqsubseteq \ell$. From this and (2), it follows that $\Delta_2(\mathtt{pc}_u) \sqsubseteq \ell$ and $\bigwedge_{v \in \mathit{vars}(q)} \Delta_2(v) \sqsubseteq \ell$. From this, it follows that $\Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in \mathit{vars}(q)} \Delta_2(v) \sqsubseteq \ell$. From this and $\Delta'_2 = \Delta_2[x \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in \mathit{vars}(q)} \Delta_2(v)]$, it follows that $\Delta'_2(\mathtt{pc}_u) \sqsubseteq \ell$, leading to a contradiction.
  - $\Delta'_1(x) \sqsubseteq \ell$, $\Delta'_2(x) \sqsubseteq \ell$, and $\Delta'_1(x) \neq \Delta'_2(x)$. From $\Delta'_1(x) \sqsubseteq \ell$ and $\Delta'_1 = \Delta_1[x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in \mathit{vars}(q)} \Delta_1(v)]$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in \mathit{vars}(q)} \Delta_1(v) \sqsubseteq \ell$. From this and (2), it follows that $\Delta_1(\mathtt{pc}_u) = \Delta_2(\mathtt{pc}_u)$ and $\bigwedge_{v \in \mathit{vars}(q)} \Delta_1(v) = \Delta_2(v)$. From this, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in \mathit{vars}(q)} \Delta_1(v) = \Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in \mathit{vars}(q)} \Delta_2(v) \sqsubseteq \ell$. From this and $\Delta'_2 = \Delta_2[x \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in \mathit{vars}(q)} \Delta_2(v)]$, it follows that $\Delta'_1(\mathtt{pc}_u) = \Delta'_2(\mathtt{pc}_u)$, leading to a contradiction.
  - $\Delta'_1(x) \sqsubseteq \ell$, $\Delta'_2(x) \sqsubseteq \ell$, and $m'_1(x) \neq m'_2(x)$. From $\Delta'_1(x) \sqsubseteq \ell$ and $\Delta'_1 = \Delta_1[x \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in \mathit{vars}(q)} \Delta_1(v)]$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \bigsqcup_{v \in \mathit{vars}(q)} \Delta_1(v) \sqsubseteq \ell$. From this, it follows that $\bigwedge_{v \in \mathit{vars}(q)} \Delta_1(v) \sqsubseteq \ell$. From this and (2), it follows that $\bigwedge_{v \in \mathit{vars}(q)} m_1(v) = m_2(v)$. From this, it follows that $q'_1 = q'_2$. From this and (1), it follows that $r_1 = r_2$. From this, $m'_1 = m_1[x \mapsto r_1]$, and $m'_2 = m_2[x \mapsto r_2]$, it follows that $m'_1(x) = m'_2(x)$, leading to a contradiction.

Induction Step. The proof of the induction step directly follows from the induction hypothesis for all rules except F-ASUSER. For the F-ASUSER rule, the proof can be done by case distinction on the executed query. The proofs for the various cases are similar to that of the rules F-SELECT, F-UPDATEDATABASEOK, and F-UPDATECONFIGURATIONOK. □

Lemma H.9 states that, under appropriate conditions, performing a step of execution in two ℓ-equivalent states with the same initial code results in the same code.

Lemma H.9. *Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, c_1, m_1, \langle s_1, \mathit{ctx}_1 \rangle \rangle$, $\langle \Delta'_1, c'_1, m'_1, \langle s'_1, \mathit{ctx}'_1 \rangle \rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, \mathit{ctx}_2 \rangle \rangle$, and $\langle \Delta'_2, c'_2, m'_2, \langle s'_2, \mathit{ctx}'_2 \rangle \rangle$ be four local configurations, and $\ell \in \mathcal{L}$ be a label. If the following conditions hold:*

1) $s_1 \equiv^{\mathit{cfg}} s_2$,
2) $\langle \Delta_1, c_1, m_1, \langle s_1, \mathit{ctx}_1 \rangle \rangle \approx_\ell \langle \Delta_2, c_2, m_2, \langle s_2, \mathit{ctx}_2 \rangle \rangle$,
3) $\Delta'_1(\mathtt{pc}_u) \sqsubseteq \ell$ *and* $\Delta'_2(\mathtt{pc}_u) \sqsubseteq \ell$,
4) $c_1 = c_2$,
5) $\langle \Delta_1, c_1, m_1, \langle s_1, \mathit{ctx}_1 \rangle \rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta'_1, c'_1, m'_1, \langle s'_1, \mathit{ctx}'_1 \rangle \rangle$,

6) $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \xrightarrow{\tau_2}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$,
then $c_1' = c_2'$.

Proof. Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle$, $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$, and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ be four local configurations, and $\ell \in \mathcal{L}$ be a label such that the following conditions hold:

1) $s_1 \equiv^{cfg} s_2$,
2) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \approx_\ell \langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$,
3) $\Delta_1'(\mathtt{pc}_u) \sqsubseteq \ell$ and $\Delta_2'(\mathtt{pc}_u) \sqsubseteq \ell$,
4) $c_1 = c_2$,
5) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \xrightarrow{\tau_1}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$,
6) $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \xrightarrow{\tau_2}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$.

We prove our claim by induction on the rules defining $\rightsquigarrow_u$.

Base Case. The proof of most of the rules, e.g., F-ASSIGN or F-OUT, is trivial. The only interesting cases are the branching statements and the expansion procedure.

- Rule F-IFTRUE. From the rule, it follows that $c_1 = \textbf{if } e \textbf{ then } c' \textbf{ else } c''$. From this and (4), it follows that $c_2 = \textbf{if } e \textbf{ then } c' \textbf{ else } c''$. From $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \xrightarrow{\tau_1}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$ and $c_1 = \textbf{if } e \textbf{ then } c' \textbf{ else } c''$, it follows that $\Delta_1' = \Delta_1[\mathtt{pc}_u \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e)]$, $m_1 = m_1'$, $\tau_1 = \epsilon$, $c_1' = c'$; $\textbf{set pc to } \Delta_1(\mathtt{pc}_u)$, and $\langle s_1, ctx_1\rangle = \langle s_1', ctx_1'\rangle$. From the rule, it also follows that $[\![e]\!](m_1) = \textbf{tt}$. From (3) and $\Delta_1' = \Delta_1[\mathtt{pc}_u \mapsto \Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e)]$, it follows that $\Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e) \sqsubseteq \ell$. From this, it follows that $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and $\Delta_1(e) \sqsubseteq \ell$. From this and (2), it follows that $\bigwedge_{v \in vars(e)} m_1(e) = m_2(e)$. From this and $[\![e]\!](m_1) = \textbf{tt}$, it follows that $[\![e]\!](m_2) = \textbf{tt}$. Therefore, by applying the rule F-IFTRUE to $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$, we obtain that $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \xrightarrow{\tau_2}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ and $c_2 = \textbf{if } e \textbf{ then } c' \textbf{ else } c''$, it follows that $\Delta_2' = \Delta_2[\mathtt{pc}_u \mapsto \Delta_2(\mathtt{pc}_u) \sqcup \Delta_2(e)]$, $m_2 = m_2'$, $\tau_2 = \epsilon$, $c_2' = c'$; $\textbf{set pc to } \Delta_2(\mathtt{pc}_u)$, and $\langle s_2, ctx_2\rangle = \langle s_2', ctx_2'\rangle$. Furthermore, from $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and (2), it follows that $\Delta_1(\mathtt{pc}_u) = \Delta_2(\mathtt{pc}_u)$. Therefore, $c_1' = c_2'$.
- Rule F-IFFALSE. The proof is similar to that for the F-IFTRUE case.
- Rule F-WHILETRUE. The proof is similar to that for the F-IFTRUE case.
- Rule F-WHILEFALSE. The proof is similar to that for the F-IFTRUE case.
- Rule F-EXPAND. From the rule, it follows that $c_1 = x \leftarrow q$. From this and (4), it follows that $c_2 = x \leftarrow q$. From $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \xrightarrow{\tau_1}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$ and $c_1 = x \leftarrow q$, it follows that $\Delta_1' = \Delta_1$, $m_1 = m_1'$, $\tau_1 = \epsilon$, $c_1' = expand(s_1, x, q, u)$, and $\langle s_1, ctx_1\rangle = \langle s_1', ctx_1'\rangle$. From $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \xrightarrow{\tau_2}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ and $c_2 = x \leftarrow q$, it follows that $\Delta_2' = \Delta_2$, $m_2 = m_2'$, $\tau_2 = \epsilon$, $c_2' = expand(s_2, x, q, u)$, and $\langle s_2, ctx_2\rangle = \langle s_2', ctx_2'\rangle$. There are a number of cases depending on q:
  - q is `SELECT` φ. For `SELECT` queries, the result of the expansion procedure is the same for any two database states s_1 and s_2. Therefore, it follows that $expand(s_1, x, q, u) = expand(s_2, x, q, u)$ and, therefore, $c_1' = c_2'$.
  - q is `INSERT` $\overline{e}$ `INTO` T. The expansion procedure relies only on the *allowed* and *apply* procedures, which, in turn, depend only on the configuration of the database state. From this and (1), it follows that $expand(s_1, x, q, u) = expand(s_2, x, q, u)$ and, therefore, $c_1' = c_2'$.
  - q is `DELETE` $\overline{e}$ `FROM` T. The proof of this case is similar to that of `INSERT` $\overline{e}$ `INTO` T.
  - q is `GRANT` p `TO` u. The expansion procedure relies only on the *allowed* and *apply* procedures, which, in turn, depend only on the configuration of the database state. From this and (1), it follows that $expand(s_1, x, q, u) = expand(s_2, x, q, u)$ and, therefore, $c_1' = c_2'$.
  - q is `GRANT` p `TO` u `WITH GRANT OPTION`. The proof of this case is similar to that of `GRANT` p `TO` u.
  - q is `REVOKE` p `FROM` u. The proof of this case is similar to that of `GRANT` p `TO` u.
  - q is a `CREATE` queries. For `CREATE` queries, the result of the expansion procedure is the same for any two database states s_1 and s_2. Therefore, it follows that $expand(s_1, x, q, u) = expand(s_2, x, q, u)$ and, therefore, $c_1' = c_2'$.
  - q is `ADD USER` u'. For `ADD USER` queries, the result of the expansion procedure is the same for any two database states s_1 and s_2. Therefore, it follows that $expand(s_1, x, q, u) = expand(s_2, x, q, u)$ and, therefore, $c_1' = c_2'$.

Induction Step. The proof of the induction step directly follows from the induction hypothesis. □

Lemma H.10 presents some results about computations involving **if** statements.

Lemma H.10. *Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle$, $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$, and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ be four local configurations, and $\ell \in \mathcal{L}$ be a label. If the following conditions hold:*

1) $s_1 \equiv^{cfg} s_2$,
2) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \approx_\ell \langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$,
3) $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ *and* $\Delta_2(\mathtt{pc}_u) \sqsubseteq \ell$,
4) $\Delta_1'(\mathtt{pc}_u) \not\sqsubseteq \ell$ *or* $\Delta_2'(\mathtt{pc}_u) \not\sqsubseteq \ell$,
5) $c_1 = c_2$,
6) $first(c_1) = \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ c'\ \textbf{\textit{else}}\ c''$,
7) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \xrightarrow{\tau_1}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$,

8) $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$,
then one of the following conditions hold:

(a) $[\![e]\!](m_1) = \mathbf{tt}$ *and for all* $\langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$ *such that* $\langle \Delta_1', c', m_1', \langle s_1', ctx_1'\rangle\rangle \overset{\tau_1'}{\rightsquigarrow}_u^{*} \langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$, $c_1'' \neq \varepsilon$,

(b) $[\![e]\!](m_1) = \mathbf{ff}$ *and for all* $\langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$ *such that* $\langle \Delta_1', c'', m_1', \langle s_1', ctx_1'\rangle\rangle \overset{\tau_1'}{\rightsquigarrow}_u^{*} \langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$, $c_1'' \neq \varepsilon$,

(c) $[\![e]\!](m_2) = \mathbf{tt}$ *and for all* $\langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$ *such that* $\langle \Delta_2', c', m_2', \langle s_2', ctx_2'\rangle\rangle \overset{\tau_2'}{\rightsquigarrow}_u^{*} \langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$, $c_2'' \neq \varepsilon$,

(d) $[\![e]\!](m_2) = \mathbf{ff}$ *and for all* $\langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$ *such that* $\langle \Delta_2', c'', m_2', \langle s_2', ctx_2'\rangle\rangle \overset{\tau_2'}{\rightsquigarrow}_u^{*} \langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$, $c_2'' \neq \varepsilon$,

(e) there are $\langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$ *and* $\langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$ *such that* $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle \overset{\tau_1'}{\rightsquigarrow}_u^{*} \langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$ *and* $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle \overset{\tau_2'}{\rightsquigarrow}_u^{*} \langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$, $c_1'' = c_2''$, *and* $\Delta_1''(\mathtt{pc}_u) \sqsubseteq \ell$ *and* $\Delta_2''(\mathtt{pc}_u) \sqsubseteq \ell$.

Proof. Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle$, $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$, and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ be four local configurations, and $\ell \in \mathcal{L}$ be a label such that:
1) $s_1 \equiv^{cfg} s_2$,
2) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \approx_\ell \langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$,
3) $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ and $\Delta_2(\mathtt{pc}_u) \sqsubseteq \ell$,
4) $\Delta_1'(\mathtt{pc}_u) \not\sqsubseteq \ell$ or $\Delta_2'(\mathtt{pc}_u) \not\sqsubseteq \ell$,
5) $c_1 = c_2$,
6) $\mathit{first}(c_1) = \mathbf{if}\ e\ \mathbf{then}\ c'\ \mathbf{else}\ c''$,
7) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$,
8) $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$,

From (5) and (6), it follows that $\mathit{first}(c_2) = \mathbf{if}\ e\ \mathbf{then}\ c'\ \mathbf{else}\ c''$. Without loss of generality, we assume that $\Delta_1'(\mathtt{pc}_u) \not\sqsubseteq \ell$. From this, $\Delta_1'(\mathtt{pc}_u) = \Delta_1(\mathtt{pc}_u) \sqcup \Delta_1(e)$. From this, (3), and $\Delta_1'(\mathtt{pc}_u) \not\sqsubseteq \ell$, it follows that $\Delta_1(e) \not\sqsubseteq \ell$. From this and (2), it follows that $\Delta_2(e) \not\sqsubseteq \ell$. From this, (3), and $\Delta_2'(\mathtt{pc}_u) = \Delta_2(\mathtt{pc}_u) \sqcup \Delta_2(e)$, it follows that $\Delta_2'(\mathtt{pc}_u) \not\sqsubseteq \ell$. Without loss of generality, we assume that $c_1 = \mathbf{if}\ e\ \mathbf{then}\ c'\ \mathbf{else}\ c''$ (the proof in case $c_1 = \mathbf{if}\ e\ \mathbf{then}\ c'\ \mathbf{else}\ c''\ ;\ c_3$ is similar). There are four cases:

- $[\![e]\!](m_1) = [\![e]\!](m_2) = \mathbf{tt}$. From the rules F-IfTrue and F-IfFalse, it follows that $c_1' = [c'\ ;\ \mathbf{set\ pc\ to}\ \Delta_1(\mathtt{pc}_u)]$ and $c_2 = [c'\ ;\ \mathbf{set\ pc\ to}\ \Delta_1(\mathtt{pc}_u)]$. There are three cases:
  - For all $\langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$ such that $\langle \Delta_1', c', m_1', \langle s_1', ctx_1'\rangle\rangle \overset{\tau_1'}{\rightsquigarrow}_u^{*} \langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$, $c_1'' \neq \varepsilon$ (i.e., c' never terminates, produces an exception, or stucks starting from $\langle \Delta_1', c', m_1', \langle s_1', ctx_1'\rangle\rangle$). In this case our claim is trivially satisfied.
  - For all $\langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$ such that $\langle \Delta_2', c', m_2', \langle s_2', ctx_2'\rangle\rangle \overset{\tau_2'}{\rightsquigarrow}_u^{*} \langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$, $c_2'' \neq \varepsilon$ (i.e., c' never terminates, produces an exception, or stucks starting from $\langle \Delta_2', c', m_2', \langle s_2', ctx_2'\rangle\rangle$). In this case our claim is trivially satisfied.
  - There exist $\langle \Delta_1'', \varepsilon, m_1'', \langle s_1'', ctx_1''\rangle\rangle$ and $\langle \Delta_2'', \varepsilon, m_2'', \langle s_2'', ctx_2''\rangle\rangle$ such that $\langle \Delta_1', c', m_1', \langle s_1', ctx_1'\rangle\rangle \overset{\tau_1'}{\rightsquigarrow}_u^{*} \langle \Delta_1'', \varepsilon, m_1'', \langle s_1'', ctx_1''\rangle\rangle$ and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle \overset{\tau_2'}{\rightsquigarrow}_u^{*} \langle \Delta_2'', \varepsilon, m_2'', \langle s_2'', ctx_2''\rangle\rangle$. From this, it follows that $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle \overset{\tau_1'}{\rightsquigarrow}_u^{*} \langle \Delta_1''', [\varepsilon\ ;\ \mathbf{set\ pc\ to}\ \Delta_1(\mathtt{pc}_u)], m_1'', \langle s_1'', ctx_1''\rangle\rangle$ and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle \overset{\tau_2'}{\rightsquigarrow}_u^{*} \langle \Delta_2''', [\varepsilon\ ;\ \mathbf{set\ pc\ to}\ \Delta_2(\mathtt{pc}_u)], m_2'', \langle s_2'', ctx_2''\rangle\rangle$. By applying the F-ExpandedCode and the F-SeqEmpty rules, we obtain $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle \overset{\tau_1'}{\rightsquigarrow}_u^{*} \langle \Delta_1''', [\mathbf{set\ pc\ to}\ \Delta_1(\mathtt{pc}_u)], m_1'', \langle s_1'', ctx_1''\rangle\rangle$ and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle \overset{\tau_2'}{\rightsquigarrow}_u^{*} \langle \Delta_2''', [\mathbf{set\ pc\ to}\ \Delta_2(\mathtt{pc}_u)], m_2'', \langle s_2'', ctx_2''\rangle\rangle$. By applying the F-ExpandedCode and the F-UpdateLabels rules, we obtain $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle \overset{\tau_1'}{\rightsquigarrow}_u^{*} \langle \Delta_1'''[\mathtt{pc}_u \mapsto \Delta_1(\mathtt{pc}_u)], [\varepsilon], m_1'', \langle s_1'', ctx_1''\rangle\rangle$ and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle \overset{\tau_2'}{\rightsquigarrow}_u^{*} \langle \Delta_2'''[\mathtt{pc}_u \mapsto \Delta_2(\mathtt{pc}_u)], [\varepsilon], m_2'', \langle s_2'', ctx_2''\rangle\rangle$ (observe that $\Delta_1'''[\mathtt{pc}_u \mapsto \Delta_1(\mathtt{pc}_u)] = \Delta_1''$ and $\Delta_2'''[\mathtt{pc}_u \mapsto \Delta_2(\mathtt{pc}_u)] = \Delta_2''$). From this, it directly follows our claim (since the code is the same in both final configurations and from (3), it follows that the $\mathtt{pc}_u$ is below ℓ in both configurations).
- $[\![e]\!](m_1) = [\![e]\!](m_2) = \mathbf{ff}$. The proof of this case is similar to that of $[\![e]\!](m_1) = [\![e]\!](m_2) = \mathbf{tt}$.
- $[\![e]\!](m_1) = \mathbf{tt}$ and $[\![e]\!](m_2) = \mathbf{ff}$. The proof of this case is similar to that of $[\![e]\!](m_1) = [\![e]\!](m_2) = \mathbf{tt}$.
- $[\![e]\!](m_1) = \mathbf{ff}$ and $[\![e]\!](m_2) = \mathbf{tt}$. The proof of this case is similar to that of $[\![e]\!](m_1) = [\![e]\!](m_2) = \mathbf{tt}$.

This completes the proof of our claim. □

Lemma H.10 presents some results about computations involving **while** statements.

Lemma H.11. *Let* sec_0 *be the policy used to initialize the monitor,* $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle$, $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$, $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$, *and* $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$ *be four local configurations, and* $\ell \in \mathcal{L}$ *be a label. If the following conditions hold:*

1) $s_1 \equiv^{cfg} s_2$,
2) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \approx_\ell \langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle$,
3) $\Delta_1(\texttt{pc}_u) \sqsubseteq \ell$ and $\Delta_2(\texttt{pc}_u) \sqsubseteq \ell$,
4) $\Delta_1'(\texttt{pc}_u) \not\sqsubseteq \ell$ or $\Delta_2'(\texttt{pc}_u) \not\sqsubseteq \ell$,
5) $c_1 = c_2$,
*6) $first(c_1) =$ **while** e **do** c',*
7) $\langle \Delta_1, c_1, m_1, \langle s_1, ctx_1\rangle\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle$,
8) $\langle \Delta_2, c_2, m_2, \langle s_2, ctx_2\rangle\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle$,
then one of the following conditions hold:

*(a) $[\![e]\!](m_1) =$ **tt** and for all $\langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$ such that $\langle \Delta_1', c', m_1', \langle s_1', ctx_1'\rangle\rangle \overset{\tau_1'}{\rightsquigarrow}_u^* \langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$, $c_1'' \neq \varepsilon$,*

*(b) $[\![e]\!](m_2) =$ **tt** and for all $\langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$ such that $\langle \Delta_2', c', m_2', \langle s_2', ctx_2'\rangle\rangle \overset{\tau_2'}{\rightsquigarrow}_u^* \langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$, $c_2'' \neq \varepsilon$,*

(c) there are $\langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$ and $\langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$ such that $\langle \Delta_1', c_1', m_1', \langle s_1', ctx_1'\rangle\rangle \overset{\tau_1'}{\rightsquigarrow}_u^* \langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$ and $\langle \Delta_2', c_2', m_2', \langle s_2', ctx_2'\rangle\rangle \overset{\tau_2'}{\rightsquigarrow}_u^* \langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$, $c_1'' = c_2''$, and $\Delta_1''(\texttt{pc}_u) \sqsubseteq \ell$ and $\Delta_2''(\texttt{pc}_u) \sqsubseteq \ell$.

Proof. The proof is similar to that of Lemma H.10. □

H.7. Lemmas about the global semantics

Here we present some auxiliary results about the global semantics of our enforcement mechanism.

Lemma H.12 states that, under appropriate conditions, performing a step of (global) execution in two ℓ-equivalent states with the same initial code and scheduler results in configurations with the same code and scheduler.

Lemma H.12. *Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, C_1, M_1, \langle s_1, ctx_1\rangle, \mathcal{S}_1\rangle$, $\langle \Delta_1', C_1', M_1', \langle s_1', ctx_1'\rangle, \mathcal{S}_1'\rangle$, $\langle \Delta_2, C_2, M_2, \langle s_2, ctx_2\rangle, \mathcal{S}_2\rangle$, and $\langle \Delta_2', C_2', M_2', \langle s_2', ctx_2'\rangle, \mathcal{S}_2'\rangle$ be four global configurations, $\ell \in \mathcal{L}$ be a label, n' be the first value in $\mathcal{S}$, $n = (n' \textbf{ mod } |C_1|) + 1$, and u be the user associated with the n-th program in C_1. If the following conditions hold:*
1) $s_1 \equiv^{cfg} s_2$,
2) $\langle \Delta_1, C_1, M_1, \langle s_1, ctx_1\rangle, \mathcal{S}_1\rangle \approx_\ell \langle \Delta_2, C_2, M_2, \langle s_2, ctx_2\rangle, \mathcal{S}_2\rangle$,
3) $\Delta_1(\texttt{pc}_u) \sqsubseteq \ell$ and $\Delta_2(\texttt{pc}_u) \sqsubseteq \ell$,
4) $\Delta_1'(\texttt{pc}_u) \sqsubseteq \ell$ and $\Delta_2'(\texttt{pc}_u) \sqsubseteq \ell$,
5) $C_1 = C_2$,
6) $\mathcal{S}_1 = \mathcal{S}_2$,
7) $\langle \Delta_1, C_1, M_1, \langle s_1, ctx_1\rangle, \mathcal{S}_1\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', C_1', M_1', \langle s_1', ctx_1'\rangle, \mathcal{S}_1'\rangle$,
8) $\langle \Delta_2, C_2, M_2, \langle s_2, ctx_2\rangle, \mathcal{S}_2\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', C_2', M_2', \langle s_2', ctx_2'\rangle, \mathcal{S}_2'\rangle$,
then $C_1' = C_2'$ and $\mathcal{S}_1' = \mathcal{S}_2'$.

Proof. The claim directly follows from (4), (5), and Lemma H.9 (together with (1) and (3)). □

Lemma H.13 states some properties of the execution of **if** statements in the global semantics.

Lemma H.13. *Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, C_1, M_1, \langle s_1, ctx_1\rangle, \mathcal{S}_1\rangle$, $\langle \Delta_1', C_1', M_1', \langle s_1', ctx_1'\rangle, \mathcal{S}_1'\rangle$, $\langle \Delta_2, C_2, M_2, \langle s_2, ctx_2\rangle, \mathcal{S}_2\rangle$, and $\langle \Delta_2', C_2', M_2', \langle s_2', ctx_2'\rangle, \mathcal{S}_2'\rangle$ be four global configurations, and $\ell \in \mathcal{L}$ be a label, n' be the first value in $\mathcal{S}$, $n = (n' \textbf{ mod } |C_1|) + 1$, and u be the user associated with the n-th program in C_1. If the following conditions hold:*
1) $s_1 \equiv^{cfg} s_2$,
2) $\langle \Delta_1, C_1, M_1, \langle s_1, ctx_1\rangle, \mathcal{S}_1\rangle \approx_\ell \langle \Delta_2, C_2, M_2, \langle s_2, ctx_2\rangle, \mathcal{S}_2\rangle$,
3) $\Delta_1(\texttt{pc}_u) \sqsubseteq \ell$ and $\Delta_2(\texttt{pc}_u) \sqsubseteq \ell$,
4) $\Delta_1'(\texttt{pc}_u) \not\sqsubseteq \ell$ or $\Delta_2'(\texttt{pc}_u) \not\sqsubseteq \ell$,
5) $C_1 = C_2$,
*6) $first(C_1|_n) =$ **if** e **then** c' **else** c'',*
7) $\mathcal{S}_1 = \mathcal{S}_2$,
8) $\langle \Delta_1, C_1, M_1, \langle s_1, ctx_1\rangle, \mathcal{S}_1\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', C_1', M_1', \langle s_1', ctx_1'\rangle, \mathcal{S}_1'\rangle$,
9) $\langle \Delta_2, C_2, M_2, \langle s_2, ctx_2\rangle, \mathcal{S}_2\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', C_2', M_2', \langle s_2', ctx_2'\rangle, \mathcal{S}_2'\rangle$,
then one of the following conditions hold:

*(a) $[\![e]\!](m_1) =$ **tt** and for all $\langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$ such that $\langle \Delta_1', c', m_1', \langle s_1', ctx_1'\rangle\rangle \overset{\tau_1'}{\rightsquigarrow}_u^* \langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$, $c_1'' \neq \varepsilon$,*

*(b) $[\![e]\!](m_1) =$ **ff** and for all $\langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$ such that $\langle \Delta_1', c'', m_1', \langle s_1', ctx_1'\rangle\rangle \overset{\tau_1'}{\rightsquigarrow}_u^* \langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$, $c_1'' \neq \varepsilon$,*

(c) $[\![e]\!](m_2) = \textbf{tt}$ *and for all* $\langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$ *such that* $\langle \Delta_2', c', m_2', \langle s_2', ctx_2'\rangle\rangle \overset{\tau_2', *}{\rightsquigarrow}_u \langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$, $c_2'' \neq \varepsilon$,

(d) $[\![e]\!](m_2) = \textbf{ff}$ *and for all* $\langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$ *such that* $\langle \Delta_2', c'', m_2', \langle s_2', ctx_2'\rangle\rangle \overset{\tau_2', *}{\rightsquigarrow}_u \langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$, $c_2'' \neq \varepsilon$,

(e) there are $\langle \Delta_1'', C_1'', M_1'', \langle s_1'', ctx_1''\rangle, \mathcal{S}_1''\rangle$ *and* $\langle \Delta_2'', C_2'', M_2'', \langle s_2'', ctx_2''\rangle, \mathcal{S}_2''\rangle$ *such that* $\langle \Delta_1', C_1', M_1', \langle s_1', ctx_1'\rangle, \mathcal{S}_1'\rangle \overset{\tau_1', *}{\rightsquigarrow}_u \langle \Delta_1'', C_1'', M_1'', \langle s_1'', ctx_1''\rangle, \mathcal{S}_1''\rangle$ *and* $\langle \Delta_2', C_2', M_2', \langle s_2', ctx_2'\rangle, \mathcal{S}_2'\rangle \overset{\tau_2', *}{\rightsquigarrow}_u \langle \Delta_2'', C_2'', M_2'', \langle s_2'', ctx_2''\rangle, \mathcal{S}_2''\rangle$, $C_1'' = C_2''$, $\mathcal{S}_1'' = \mathcal{S}_2''$, *and* $\Delta_1''(\mathtt{pc}_u) \sqsubseteq \ell$ *and* $\Delta_2''(\mathtt{pc}_u) \sqsubseteq \ell$.

Proof. From (6) and the rules F-IFTRUE and F-IFFALSE, it follows that the only applicable rule in the global semantics is F-ATOMIC-STATEMENT. Our claim directly follows from this, (3), (4), Lemma H.10, and the fact that the F-ATOMIC-STATEMENT rule does not modify the scheduler. □

Lemma H.14 states some properties of the execution of **while** statements in the global semantics.

Lemma H.14. *Let sec_0 be the policy used to initialize the monitor, $\langle \Delta_1, C_1, M_1, \langle s_1, ctx_1\rangle, \mathcal{S}_1\rangle$, $\langle \Delta_1', C_1', M_1', \langle s_1', ctx_1'\rangle, \mathcal{S}_1'\rangle$, $\langle \Delta_2, C_2, M_2, \langle s_2, ctx_2\rangle, \mathcal{S}_2\rangle$, and $\langle \Delta_2', C_2', M_2', \langle s_2', ctx_2'\rangle, \mathcal{S}_2'\rangle$ be four global configurations, and $\ell \in \mathcal{L}$ be a label, n' be the first value in $\mathcal{S}$, $n = (n' \bmod |C_1|) + 1$, and u be the user associated with the n-th program in C_1. If the following conditions hold:*

1) $s_1 \equiv^{cfg} s_2$,
2) $\langle \Delta_1, C_1, M_1, \langle s_1, ctx_1\rangle, \mathcal{S}_1\rangle \approx_\ell \langle \Delta_2, C_2, M_2, \langle s_2, ctx_2\rangle, \mathcal{S}_2\rangle$,
3) $\Delta_1(\mathtt{pc}_u) \sqsubseteq \ell$ *and* $\Delta_2(\mathtt{pc}_u) \sqsubseteq \ell$,
4) $\Delta_1'(\mathtt{pc}_u) \not\sqsubseteq \ell$ *or* $\Delta_2'(\mathtt{pc}_u) \not\sqsubseteq \ell$,
5) $C_1 = C_2$,
6) $first(C_1|_n) = \textbf{while } e \textbf{ do } c'$,
7) $\mathcal{S}_1 = \mathcal{S}_2$,
8) $\langle \Delta_1, C_1, M_1, \langle s_1, ctx_1\rangle, \mathcal{S}_1\rangle \overset{\tau_1}{\rightsquigarrow}_u \langle \Delta_1', C_1', M_1', \langle s_1', ctx_1'\rangle, \mathcal{S}_1'\rangle$,
9) $\langle \Delta_2, C_2, M_2, \langle s_2, ctx_2\rangle, \mathcal{S}_2\rangle \overset{\tau_2}{\rightsquigarrow}_u \langle \Delta_2', C_2', M_2', \langle s_2', ctx_2'\rangle, \mathcal{S}_2'\rangle$,

then one of the following conditions hold:

(a) $[\![e]\!](m_1) = \textbf{tt}$ *and for all* $\langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$ *such that* $\langle \Delta_1', c', m_1', \langle s_1', ctx_1'\rangle\rangle \overset{\tau_1', *}{\rightsquigarrow}_u \langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$, $c_1'' \neq \varepsilon$,

(b) $[\![e]\!](m_1) = \textbf{ff}$ *and for all* $\langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$ *such that* $\langle \Delta_1', c'', m_1', \langle s_1', ctx_1'\rangle\rangle \overset{\tau_1', *}{\rightsquigarrow}_u \langle \Delta_1'', c_1'', m_1'', \langle s_1'', ctx_1''\rangle\rangle$, $c_1'' \neq \varepsilon$,

(c) $[\![e]\!](m_2) = \textbf{tt}$ *and for all* $\langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$ *such that* $\langle \Delta_2', c', m_2', \langle s_2', ctx_2'\rangle\rangle \overset{\tau_2', *}{\rightsquigarrow}_u \langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$, $c_2'' \neq \varepsilon$,

(d) $[\![e]\!](m_2) = \textbf{ff}$ *and for all* $\langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$ *such that* $\langle \Delta_2', c'', m_2', \langle s_2', ctx_2'\rangle\rangle \overset{\tau_2', *}{\rightsquigarrow}_u \langle \Delta_2'', c_2'', m_2'', \langle s_2'', ctx_2''\rangle\rangle$, $c_2'' \neq \varepsilon$,

(e) there are $\langle \Delta_1'', C_1'', M_1'', \langle s_1'', ctx_1''\rangle, \mathcal{S}_1''\rangle$ *and* $\langle \Delta_2'', C_2'', M_2'', \langle s_2'', ctx_2''\rangle, \mathcal{S}_2''\rangle$ *such that* $\langle \Delta_1', C_1', M_1', \langle s_1', ctx_1'\rangle, \mathcal{S}_1'\rangle \overset{\tau_1', *}{\rightsquigarrow}_u \langle \Delta_1'', C_1'', M_1'', \langle s_1'', ctx_1''\rangle, \mathcal{S}_1''\rangle$ *and* $\langle \Delta_2', C_2', M_2', \langle s_2', ctx_2'\rangle, \mathcal{S}_2'\rangle \overset{\tau_2', *}{\rightsquigarrow}_u \langle \Delta_2'', C_2'', M_2'', \langle s_2'', ctx_2''\rangle, \mathcal{S}_2''\rangle$, $C_1'' = C_2''$, $\mathcal{S}_1'' = \mathcal{S}_2''$, *and* $\Delta_1''(\mathtt{pc}_u) \sqsubseteq \ell$ *and* $\Delta_2''(\mathtt{pc}_u) \sqsubseteq \ell$.

Proof. From (6) and the rules F-WHILETRUE and F-WHILEFALSE, it follows that the only applicable rule in the global semantics is F-ATOMIC-STATEMENT. Our claim directly follows from this, (3), (4), Lemma H.11, and the fact that the F-ATOMIC-STATEMENT rule does not modify the schedulers. □

H.8. Bisimulations

Here we introduce bisimulations for our setting, and we prove some key results about them.

We first introduce some machinery. Let $\sigma_1 = \langle \Delta_1, C_1, M_1, \langle s_1, ctx_1\rangle, \mathcal{S}_1\rangle$ and $\sigma_2 = \langle \Delta_2, C_2, M_2, \langle s_2, ctx_2\rangle, \mathcal{S}_2\rangle$ be two global configurations and $i \in \mathbb{N}$ be an integer. We say that σ_2 is reachable in at most i steps from σ_1, denoted $reach^i(\sigma_1, \sigma_2)$, iff there exists an $i' \leq i$ such that $\sigma_1 \overset{\tau, i'}{\rightsquigarrow} \sigma_2$. The current program in σ_1, denoted $currPrg(\sigma_1)$, is c and the current memory $currMem(\sigma_1)$ is m, where n' is the first element in $\mathcal{S}_1$, $n = (n' \,\mathbf{mod} |C_1|) + 1$, $C_1|_n = \langle u, c\rangle$, and $M|_n = \langle u, c\rangle$. Furthermore, the current pc in σ_1, denoted $currPc(\sigma_1)$, is $\Delta_1(\mathtt{pc}_u)$, where n' is the first element in $\mathcal{S}_1$, $n = (n' \,\mathbf{mod} |C_1|) + 1$, and $C_1|_n = \langle u, c\rangle$. Finally, the current user in σ_1, denoted $currUsr(\sigma_1)$, is u, where $currPrg(\sigma_1) = \langle u, c\rangle$. Given a local configuration σ and a user u, $term(\sigma, u) = \top$ iff there exists a $\langle \Delta, c, m, \langle s, ctx\rangle\rangle$ such that $\sigma \overset{\tau, *}{\rightsquigarrow}_u \langle \Delta, c, m, \langle s, ctx\rangle\rangle$ and $c = \varepsilon$. Given a label ℓ and a user u, we denote by $notBelow(\sigma, \ell, u) = \top$ iff for all $\langle \Delta, c, m, \langle s, ctx\rangle\rangle$ such that $\sigma \overset{\tau, *}{\rightsquigarrow}_u \langle \Delta, c, m, \langle s, ctx\rangle\rangle$, then $\Delta(\mathtt{pc}_u) \not\sqsubseteq \ell$.

We are now ready to formalize bisimulations.

Definition 5. Let $\sigma_1 = \langle \Delta_1, C_1, M_1, \langle s_1, ctx_1\rangle, \mathcal{S}_1\rangle$ and $\sigma_2 = \langle \Delta_2, C_2, M_2, \langle s_2, ctx_2\rangle, \mathcal{S}_2\rangle$ be two global configurations, $i, j \in \mathbb{N}$ be integers, and $\ell \in \mathcal{L}$ be a label. Furthermore, let R be a binary relation over global configurations. We say that

R is a $(\sigma_1, \sigma_2, i, j, \ell)$-*bisimulation* iff for all $\sigma_1' = \langle \Delta_1', C_1', M_1', \langle s_1', ctx_1' \rangle, \mathcal{S}_1' \rangle$ and $\sigma_2' = \langle \Delta_2', C_2', M_2', \langle s_2', ctx_2' \rangle, \mathcal{S}_2' \rangle$ such that $\sigma_1' R \sigma_2'$, then the following conditions hold:

1) $reach^i(\sigma_1, \sigma_1')$ and $reach^j(\sigma_2, \sigma_2')$.
2) $\sigma_1' \approx_\ell \sigma_2'$.
3) $\sigma_1' \equiv^{cfg} \sigma_2'$.
4) $C_1' = C_2'$.
5) $\mathcal{S}_1' = \mathcal{S}_2'$.
6) $currPc(\sigma_1) \sqsubseteq \ell$ and $currPc(\sigma_2) \sqsubseteq \ell$.
7) If $\sigma_1' \overset{\tau_1'}{\rightsquigarrow} \sigma_1''$, $\sigma_2' \overset{\tau_2'}{\rightsquigarrow} \sigma_2''$, $reach^i(\sigma_1, \sigma_1'')$, $reach^j(\sigma_2, \sigma_2'')$, and $currPc(\sigma_1'') \sqsubseteq \ell \wedge currPc(\sigma_2'') \sqsubseteq \ell$, then $\sigma_1'' R \sigma_2''$.
8) If $\sigma_1' \overset{\tau_1'}{\rightsquigarrow} \sigma_1''$, $\sigma_2' \overset{\tau_2'}{\rightsquigarrow} \sigma_2''$, $reach^i(\sigma_1, \sigma_1'')$, $reach^j(\sigma_2, \sigma_2'')$, and $currPc(\sigma_1'') \not\sqsubseteq \ell \vee currPc(\sigma_2'') \not\sqsubseteq \ell$, then one of the following conditions hold:
   a) for all σ_1'' and σ_2'' such that $\sigma_1' \overset{\tau_1''}{\rightsquigarrow}{}^{*} \sigma_1''$, $\sigma_2' \overset{\tau_2''}{\rightsquigarrow}{}^{*} \sigma_2''$, $reach^i(\sigma_1, \sigma_1'')$, $reach^j(\sigma_2, \sigma_2'')$, $currPc(\sigma_1) \not\sqsubseteq \ell$ or $currPc(\sigma_2) \not\sqsubseteq \ell$, or
   b) there are σ_1'' and σ_2'' such that $\sigma_1' \overset{\tau_1''}{\rightsquigarrow}{}^{*} \sigma_1''$, $\sigma_2' \overset{\tau_2''}{\rightsquigarrow}{}^{*} \sigma_2''$, $reach^i(\sigma_1, \sigma_1'')$, $reach^j(\sigma_2, \sigma_2'')$, and $\sigma_1'' R \sigma_2''$. □

Lemmas H.15 and H.16 state that, under certain conditions, we can construct bisimulations.

Lemma H.15. *Let sec_0 be the policy used to initialize the monitor, $\sigma_1^0 = \langle \Delta_1^0, C_1^0, M_1^0, \langle s_1^0, ctx_1^0 \rangle, \mathcal{S}_1^0 \rangle$, $\sigma_2^0 = \langle \Delta_2^0, C_2^0, M_2^0, \langle s_2^0, ctx_2^0 \rangle, \mathcal{S}_2^0 \rangle$, $\sigma_1^1 = \langle \Delta_1^1, C_1^1, M_1^1, \langle s_1^1, ctx_1^1 \rangle, \mathcal{S}_1^1 \rangle$, $\sigma_2^1 = \langle \Delta_2^1, C_2^1, M_2^1, \langle s_2^1, ctx_2^1 \rangle, \mathcal{S}_2^1 \rangle$ be four global configurations, τ_1, τ_2 be two traces, u be a user, and $\ell \in \mathcal{L}$ be a label. If the following conditions hold:*

1) $\sigma_1^0 \overset{\tau_1}{\rightsquigarrow} \sigma_1^1$,
2) $\sigma_2^0 \overset{\tau_2}{\rightsquigarrow} \sigma_2^1$,
3) $\sigma_1^0 \approx_\ell \sigma_2^0$,
4) $\sigma_1^0 \equiv^{cfg} \sigma_2^0$,
5) $C_1^0 = C_2^0$,
6) $\mathcal{S}_1^0 = \mathcal{S}_2^0$,
7) $currPc(\sigma_1^0) \sqsubseteq \ell$ and $currPc(\sigma_2^0) \sqsubseteq \ell$,
8) $currPc(\sigma_1^1) \sqsubseteq \ell$ and $currPc(\sigma_2^1) \sqsubseteq \ell$,
9) $cl(auth(sec_0, atk)) \sqsubseteq \ell$,

then $\{(\sigma_1^0, \sigma_2^0), (\sigma_1^1, \sigma_2^1)\}$ is a $(\sigma_1^0, \sigma_2^0, 1, 1, \ell)$-bisimulation.

Proof. Let sec_0 be the policy used to initialize the monitor, $\sigma_1^0 = \langle \Delta_1^0, C_1^0, M_1^0, \langle s_1^0, ctx_1^0 \rangle, \mathcal{S}_1^0 \rangle$, $\sigma_2^0 = \langle \Delta_2^0, C_2^0, M_2^0, \langle s_2^0, ctx_2^0 \rangle, \mathcal{S}_2^0 \rangle$, $\sigma_1^1 = \langle \Delta_1^1, C_1^1, M_1^1, \langle s_1^1, ctx_1^1 \rangle, \mathcal{S}_1^1 \rangle$, $\sigma_2^1 = \langle \Delta_2^1, C_2^1, M_2^1, \langle s_2^1, ctx_2^1 \rangle, \mathcal{S}_2^1 \rangle$ be four global configurations, τ_1, τ_2 be two traces, u be a user, and $\ell \in \mathcal{L}$ be a label. Furthermore, we assume that following conditions hold:

1) $\sigma_1^0 \overset{\tau_1}{\rightsquigarrow} \sigma_1^1$,
2) $\sigma_2^0 \overset{\tau_2}{\rightsquigarrow} \sigma_2^1$,
3) $\sigma_1^0 \approx_\ell \sigma_2^0$,
4) $\sigma_1^0 \equiv^{cfg} \sigma_2^0$,
5) $C_1^0 = C_2^0$,
6) $\mathcal{S}_1^0 = \mathcal{S}_2^0$,
7) $currPc(\sigma_1^0) \sqsubseteq \ell$ and $currPc(\sigma_2^0) \sqsubseteq \ell$,
8) $currPc(\sigma_1^1) \sqsubseteq \ell$ and $currPc(\sigma_2^1) \sqsubseteq \ell$,
9) $cl(auth(sec_0, atk)) \sqsubseteq \ell$.

We now show that $\{(\sigma_1^0, \sigma_2^0), (\sigma_1^1, \sigma_2^1)\}$ is a $(\sigma_1^0, \sigma_2^0, 1, 1, \ell)$-bisimulation. We first need to show that for all $\sigma_1' = \langle \Delta_1', C_1', M_1', \langle s_1', ctx_1' \rangle, \mathcal{S}_1' \rangle$ and $\sigma_2' = \langle \Delta_2', C_2', M_2', \langle s_2', ctx_2' \rangle, \mathcal{S}_2' \rangle$ such that $\sigma_1' R \sigma_2'$, the following conditions hold: (a) $reach^1(\sigma_1^0, \sigma_1')$ and $reach^1(\sigma_2^0, \sigma_2')$, (b) $\sigma_1' \approx_\ell \sigma_2'$, (c) $\sigma_1' \equiv^{cfg} \sigma_2'$, (d) $C_1' = C_2'$, (e) $\mathcal{S}_1' = \mathcal{S}_2'$, (f) $currPc(\sigma_1) \sqsubseteq \ell$ and $currPc(\sigma_2) \sqsubseteq \ell$. There are two cases:

- $(\sigma_1', \sigma_2') = (\sigma_1^0, \sigma_2^0)$. Then, (a) trivially follows since $reach^1(\sigma, \sigma)$ always holds. Moreover, (b)–(f) directly follow from (3)–(7).
- $(\sigma_1', \sigma_2') = (\sigma_1^1, \sigma_2^1)$. Then, (a) directly follows from (1) and (2). There are two cases:
  - (1) is obtained by applying the M-EVAL-END rule. From this and (5), it follows that also (2) is obtained using the M-EVAL-END rule. From this, (5), and the rule, we eliminate in both run the same components. From this and (3)–(7), it directly follows that (b)–(f) are satisfied.
  - (1) is obtained by applying the M-EVAL-STEP or M-ATOMIC-STATEMENT rules. From this, (5), and (6), it follows that we perform one step of the local semantics for the same program in both runs. From this and (6), (e) directly follows. From (3)–(10), Lemmas H.7, H.8, and H.9, conditions (b)–(d) follow. Finally, condition (f) immediately follows from (8).

Therefore, the fact that R is a bisimulation directly follows from (i) the fact that (a)–(f) hold for (σ_1^0, σ_2^0) and (σ_1^1, σ_2^1), (ii) assumptions (1), (2), and $R = \{(\sigma_1^0, \sigma_2^0), (\sigma_1^1, \sigma_2^1)\}$, and (3) there are no configurations that are reachable in 1 step from σ_1^0 and σ_2^0 other than σ_1^1 and σ_2^1. □

Lemma H.16. *Let sec_0 be the policy used to initialize the monitor, $\sigma_1^0 = \langle \Delta_1^0, C_1^0, M_1^0, \langle s_1^0, ctx_1^0 \rangle, \mathcal{S}_1^0 \rangle$, $\sigma_2^0 = \langle \Delta_2^0, C_2^0, M_2^0, \langle s_2^0, ctx_2^0 \rangle, \mathcal{S}_2^0 \rangle$, $\sigma_1^1 = \langle \Delta_1^1, C_1^1, M_1^1, \langle s_1^1, ctx_1^1 \rangle, \mathcal{S}_1^1 \rangle$, $\sigma_2^1 = \langle \Delta_2^1, C_2^1, M_2^1, \langle s_2^1, ctx_2^1 \rangle, \mathcal{S}_2^1 \rangle$ be four global configurations, τ_1, τ_2 be two traces, $u = currUsr(\sigma_1^1)$ be a user, and $\ell \in \mathcal{L}$ be a label. If the following conditions hold:*

1) *$\sigma_1^0 \overset{\tau_1}{\rightsquigarrow} \sigma_1^1$,*
2) *$\sigma_2^0 \overset{\tau_2}{\rightsquigarrow} \sigma_2^1$,*
3) *$\sigma_1^0 \approx_\ell \sigma_2^0$,*
4) *$\sigma_1^0 \equiv^{cfg} \sigma_2^0$,*
5) *$C_1^0 = C_2^0$,*
6) *$\mathcal{S}_1^0 = \mathcal{S}_2^0$,*
7) *$currPc(\sigma_1^0) \sqsubseteq \ell$ and $currPc(\sigma_2^0) \sqsubseteq \ell$,*
8) *$currPc(\sigma_1^1) \not\sqsubseteq \ell$ or $currPc(\sigma_2^1) \not\sqsubseteq \ell$,*
9) *$cl(auth(sec_0, atk)) \sqsubseteq \ell$,*
10) *for all users $u' \neq currUsr(\sigma_1^0)$, $\Delta_1^0(\mathtt{pc}_{u'}) \sqsubseteq \ell$ and $\Delta_2^0(\mathtt{pc}_{u'}) \sqsubseteq \ell$,*
11) *whenever $first(currPrg(\sigma_1^0)) =$ **if** e **then** c' **else** c'' and $[\![e]\!](M_1^0) =$ **tt**, then $term(\langle \Delta_1^1, c', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1 \rangle \rangle, u) = \top$ and $notBelow(\langle \Delta_1^1, c', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1 \rangle \rangle, \ell, u) = \top$,*
12) *whenever $first(currPrg(\sigma_1^0)) =$ **if** e **then** c' **else** c'' and $[\![e]\!](M_1^0) =$ **ff**, then $term(\langle \Delta_1^1, c'', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1 \rangle \rangle, u) = \top$ and $notBelow(\langle \Delta_1^1, c'', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1 \rangle \rangle, \ell, u) = \top$,*
13) *whenever $first(currPrg(\sigma_2^0)) =$ **if** e **then** c' **else** c'' and $[\![e]\!](M_2^0) =$ **tt**, then $term(\langle \Delta_2^1, c', currMem(\sigma_2^1), \langle s_2^1, ctx_2^1 \rangle \rangle, u) = \top$ and $notBelow(\langle \Delta_2^1, c', currMem(\sigma_2^1), \langle s_2^1, ctx_2^1 \rangle \rangle, \ell, u) = \top$,*
14) *whenever $first(currPrg(\sigma_2^0)) =$ **if** e **then** c' **else** c'' and $[\![e]\!](M_2^0) =$ **ff**, then $term(\langle \Delta_2^1, c'', currMem(\sigma_2^1), \langle s_2^1, ctx_2^1 \rangle \rangle, u) = \top$ and $notBelow(\langle \Delta_2^1, c'', currMem(\sigma_2^1), \langle s_2^1, ctx_2^1 \rangle \rangle, \ell, u) = \top$,*
15) *whenever $first(currPrg(\sigma_1^0)) =$ **while** e **do** c' and $[\![e]\!](M_1^0) =$ **tt**, then $term(\langle \Delta_1^1, c';$**while** e **do** $c', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1 \rangle \rangle, u) = \top$ and $notBelow(\langle \Delta_1^1, c', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1 \rangle \rangle, \ell, u) = \top$,*
16) *whenever $first(currPrg(\sigma_2^0)) =$ **while** e **do** c' and $[\![e]\!](M_2^0) =$ **tt**, then $term(\langle \Delta_2^1, c';$**while** e **do** $c', currMem(\sigma_2^1), \langle s_2^1, ctx_2^1 \rangle \rangle, u) = \top$ and $notBelow(\langle \Delta_2^1, c', currMem(\sigma_2^1), \langle s_2^1, ctx_2^1 \rangle \rangle, \ell, u) = \top$,*
17) *whenever $first(currPrg(\sigma_1^0)) =$ **set pc to** ℓ', $\ell' \sqsubseteq currPc(\sigma_1^0)$,*

then there are i, j such that $\sigma_1^0 \overset{\pi_1}{\rightsquigarrow}{}^{i} \sigma_1^i$, $\sigma_2^0 \overset{\pi_2}{\rightsquigarrow}{}^{j} \sigma_2^j$, and $\{(\sigma_1^0, \sigma_2^0), (\sigma_1^i, \sigma_2^j)\}$ is a $(\sigma_1^0, \sigma_2^0, i, j, \ell)$-bisimulation.

Proof. Let sec_0 be the policy used to initialize the monitor, $\sigma_1^0 = \langle \Delta_1^0, C_1^0, M_1^0, \langle s_1^0, ctx_1^0 \rangle, \mathcal{S}_1^0 \rangle$, $\sigma_2^0 = \langle \Delta_2^0, C_2^0, M_2^0, \langle s_2^0, ctx_2^0 \rangle, \mathcal{S}_2^0 \rangle$, $\sigma_1^1 = \langle \Delta_1^1, C_1^1, M_1^1, \langle s_1^1, ctx_1^1 \rangle, \mathcal{S}_1^1 \rangle$, $\sigma_2^1 = \langle \Delta_2^1, C_2^1, M_2^1, \langle s_2^1, ctx_2^1 \rangle, \mathcal{S}_2^1 \rangle$ be four global configurations, τ_1, τ_2 be two traces, $u = currUsr(\sigma_1^1)$ be a user, and $\ell \in \mathcal{L}$ be a label. Furthermore, we assume the following conditions hold:

1) $\sigma_1^0 \overset{\tau_1}{\rightsquigarrow} \sigma_1^1$,
2) $\sigma_2^0 \overset{\tau_2}{\rightsquigarrow} \sigma_2^1$,
3) $\sigma_1^0 \approx_\ell \sigma_2^0$,
4) $\sigma_1^0 \equiv^{cfg} \sigma_2^0$,
5) $C_1^0 = C_2^0$,
6) $\mathcal{S}_1^0 = \mathcal{S}_2^0$,
7) $currPc(\sigma_1^0) \sqsubseteq \ell$ and $currPc(\sigma_2^0) \sqsubseteq \ell$,
8) $currPc(\sigma_1^1) \not\sqsubseteq \ell$ or $currPc(\sigma_2^1) \not\sqsubseteq \ell$,
9) $cl(auth(sec_0, atk)) \sqsubseteq \ell$,
10) for all users $u' \neq currUsr(\sigma_1^0)$, $\Delta_1^0(\mathtt{pc}_{u'}) \sqsubseteq \ell$ and $\Delta_2^0(\mathtt{pc}_{u'}) \sqsubseteq \ell$,
11) whenever $first(currPrg(\sigma_1^0)) =$ **if** e **then** c' **else** c'' and $[\![e]\!](M_1^0) =$ **tt**, then $term(\langle \Delta_1^1, c', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1 \rangle \rangle, u) = \top$ and $notBelow(\langle \Delta_1^1, c', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1 \rangle \rangle, \ell, u) = \top$,
12) whenever $first(currPrg(\sigma_1^0)) =$ **if** e **then** c' **else** c'' and $[\![e]\!](M_1^0) =$ **ff**, then $term(\langle \Delta_1^1, c'', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1 \rangle \rangle, u) = \top$ and $notBelow(\langle \Delta_1^1, c'', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1 \rangle \rangle, \ell, u) = \top$,
13) whenever $first(currPrg(\sigma_2^0)) =$ **if** e **then** c' **else** c'' and $[\![e]\!](M_2^0) =$ **tt**, then $term(\langle \Delta_2^1, c', currMem(\sigma_2^1), \langle s_2^1, ctx_2^1 \rangle \rangle, u) = \top$ and $notBelow(\langle \Delta_2^1, c', currMem(\sigma_2^1), \langle s_2^1, ctx_2^1 \rangle \rangle, \ell, u) = \top$,
14) whenever $first(currPrg(\sigma_2^0)) =$ **if** e **then** c' **else** c'' and $[\![e]\!](M_2^0) =$ **ff**, then $term(\langle \Delta_2^1, c'', currMem(\sigma_2^1), \langle s_2^1, ctx_2^1 \rangle \rangle, u) = \top$ and $notBelow(\langle \Delta_2^1, c'', currMem(\sigma_2^1), \langle s_2^1, ctx_2^1 \rangle \rangle, \ell, u) = \top$,
15) whenever $first(currPrg(\sigma_1^0)) =$ **while** e **do** c' and $[\![e]\!](M_1^0) =$ **tt**, then $term(\langle \Delta_1^1, c';$**while** e **do** $c', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1 \rangle \rangle, u) = \top$ and $notBelow(\langle \Delta_1^1, c', currMem(\sigma_1^1), \langle s_1^1, ctx_1^1 \rangle \rangle, \ell, u) = \top$,
16) whenever $first(currPrg(\sigma_2^0)) =$ **while** e **do** c' and $[\![e]\!](M_2^0) =$ **tt**, then $term(\langle \Delta_2^1, c';$**while** e **do** $c', currMem(\sigma_2^1), \langle s_2^1, ctx_2^1 \rangle \rangle, u) = \top$ and $notBelow(\langle \Delta_2^1, c', currMem(\sigma_2^1), \langle s_2^1, ctx_2^1 \rangle \rangle, \ell, u) = \top$,
17) whenever $first(currPrg(\sigma_1^0)) =$ **set pc to** ℓ', $\ell' \sqsubseteq currPc(\sigma_1^0)$.

Let $c_1 = currPrg(\sigma_1^0)$, $u_1 = currUsr(\sigma_1^0)$, $c_2 = currPrg(\sigma_2^0)$, and $u_2 = currUsr(\sigma_2^0)$. From (5) and (6), it follows that $\langle u_1, c_1 \rangle = \langle u_2, c_2 \rangle$. In the following, we denote u_1 and u_2 using u and c_1 and c_2 using c. Furthermore, we denote by n the value such that $C_1^0|_n = \langle currUsr(\sigma_1^0), currPrg(\sigma_1^0) \rangle$. From Lemma H.2 and (17), there are only two cases:

- $first(c_1) =$ **if** e **then** c' **else** c''. We assume that $c_1 =$ **if** e **then** c' **else** $c''; c_3$, where c_3 can be an empty program (the proof for the other cases is almost identical). From this and the F-IfTrue and F-IfFalse rules, it follows that $currPrg(\sigma_1^1) = [c_1^*;$**set pc to** $\Delta_1^0(\mathtt{pc}_u)]; c_3$ and $currPrg(\sigma_2^1) = [c_2^*;$**set pc to** $\Delta_2^0(\mathtt{pc}_u)]; c_3$, where $c_1^* \in \{c', c''\}$, and $c_2^* \in \{c', c''\}$. Furthermore, from (3) and (7), it follows that $\Delta_1^0(\mathtt{pc}_u) = \Delta_2^0(\mathtt{pc}_u) = \ell'$. Therefore, $currPrg(\sigma_1^1) = [c_1^*;$**set pc to** $\ell']; c_3$ and $currPrg(\sigma_2^1) = [c_2^*;$**set pc to** $\ell']; c_3$, where $c_1^* \in \{c', c''\}$, and $c_2^* \in \{c', c''\}$.

From this and (11)–(14), it follows that there are $\langle \Delta_1^i, [\varepsilon]; c_3, m_1^i, \langle s_1^i, ctx_1^i \rangle\rangle$ and $\langle \Delta_2^j, [\varepsilon]; c_3, m_2^j, \langle s_2^j, ctx_2^j \rangle\rangle$ such that $\langle \Delta_1^0, [c_1^*; \textbf{set pc to } \ell']; c_3, currMem(\sigma_1^0), \langle s_1^0, ctx_1^0 \rangle\rangle \overset{\tau}{\rightsquigarrow}{}^{i}_{u} \langle \Delta_1^i, \varepsilon; c_3, m_1^i, \langle s_1^i, ctx_1^i \rangle\rangle$ and $\langle \Delta_2^0, [c_2^*; \textbf{set pc to } \ell']; c_3,$ $currMem(\sigma_2^0), \langle s_2^0, ctx_2^0 \rangle\rangle \overset{\tau'}{\rightsquigarrow}{}^{j}_{u} \langle \Delta_2^j, \varepsilon; c_3, m_2^j, \langle s_2^j, ctx_2^j \rangle\rangle$, where $\Delta_1^i(\mathtt{pc}_u) = \ell'$ and $\Delta_2^i(\mathtt{pc}_u) = \ell'$. From (11)–(14), it follows that during both computations $\mathtt{pc}_u$ is never below ℓ before executing the last **set pc to** ℓ' statement. From this and (8), it follows that $\Delta_1^{i'}(\mathtt{pc}_u) \not\sqsubseteq \ell$ and $\Delta_2^{j'}(\mathtt{pc}_u) \not\sqsubseteq \ell$ for all $1 \leq i' \leq i-1$ and $1 \leq j' \leq j-1$. By repeatedly applying Lemma H.3 and Lemma H.4 and $\Delta_1^{i'}(\mathtt{pc}_\mu) \not\sqsubseteq \ell$ and $\Delta_2^{j'}(\mathtt{pc}_u) \not\sqsubseteq \ell$ for all $1 \leq i' \leq i-1$ and $1 \leq j' \leq j-1$, we obtain that $\langle \Delta_1^{i'}, c_1^{i'}, m_1^{i'}, \langle s_1^{i'}, ctx_1^{i'} \rangle\rangle \approx_\ell \langle \Delta_2^{j'}, c_2^{j'}, m_2^{j'}, \langle s_2^{j'}, ctx_2^{j'} \rangle\rangle$ for all $1 \leq i' \leq i-1$ and $1 \leq j' \leq j-1$. From this and the fact that in the last step of the execution we set $\mathtt{pc}_u$ to ℓ' in both runs, $\langle \Delta_1^i, c_1^i, m_1^i, \langle s_1^i, ctx_1^i \rangle\rangle \approx_\ell$ $\langle \Delta_2^j, c_2^j, m_2^j, \langle s_2^j, ctx_2^j \rangle\rangle$. Similarly, by repeatedly applying Lemma H.5, we obtain that $\langle \Delta_1^i, c_1^i, m_1^i, \langle s_1^i, ctx_1^i \rangle\rangle \equiv^{cfg}$ $\langle \Delta_2^j, c_2^j, m_2^j, \langle s_2^j, ctx_2^j \rangle\rangle$. By repeatedly applying the F-ATOMIC-STATEMENT rule both to σ_1^0 and σ_2^0, we obtain that $\sigma_1^0 \overset{\tau}{\rightsquigarrow}{}^{i} \langle \Delta_1^i, C_1^i, M_1^i, \langle s_1^i, ctx_1^i \rangle, \mathcal{S}_1^0 \rangle$ and $\sigma_2^0 \overset{\tau'}{\rightsquigarrow}{}^{i} \langle \Delta_2^j, C_2^j, M_2^j, \langle s_2^j, ctx_2^j \rangle, \mathcal{S}_2^0 \rangle$, where $C_1^i = C_1^i|_1 \cdot \ldots \cdot C_1^i|_{n-1} \cdot \langle u,$ $[\varepsilon]; c_3 \rangle \cdot C_1^i|_{n+1} \ldots \cdot C_1^i|_{|C_1^i|}$, $C_2^i = C_2^i|_1 \cdot \ldots \cdot C_2^i|_{n-1} \cdot \langle u, [\varepsilon]; c_3 \rangle \cdot C_2^i|_{n+1} \ldots \cdot C_2^i|_{|C_2^i|}$, $M_1^i = M_1^i|_1 \cdot \ldots \cdot M_1^i|_{n-1} \cdot \langle u,$ $m_1^i \rangle \cdot M_1^i|_{n+1} \ldots \cdot M_1^i|_{|M_1^i|}$, and $M_2^i = M_2^i|_1 \cdot \ldots \cdot M_2^i|_{n-1} \cdot \langle u, m_2^i \rangle \cdot M_2^i|_{n+1} \ldots \cdot M_2^i|_{|M_2^i|}$. In the following, let $\sigma_1^i =$ $\langle \Delta_1^i, C_1^i, M_1^i, \langle s_1^i, ctx_1^i \rangle, \mathcal{S}_1^0 \rangle$ and $\sigma_2^j = \langle \Delta_2^j, C_2^j, M_2^j, \langle s_2^j, ctx_2^j \rangle, \mathcal{S}_2^0 \rangle$. We now show that $R = \{(\sigma_1^0, \sigma_2^0), (\sigma_1^i, \sigma_2^j)\}$ is a $(\sigma_1^0, \sigma_2^0, i, j, \ell)$-bisimulation. We first need to show that for all $\sigma_1' = \langle \Delta_1', C_1', M_1', \langle s_1', ctx_1' \rangle, \mathcal{S}_1' \rangle$ and $\sigma_2' = \langle \Delta_2', C_2',$ $M_2', \langle s_2', ctx_2' \rangle, \mathcal{S}_2' \rangle$ such that $\sigma_1' R \sigma_2'$, the following conditions hold: (a) $reach^i(\sigma_1^0, \sigma_1')$ and $reach^j(\sigma_2^0, \sigma_2')$, (b) $\sigma_1' \approx_\ell$ σ_2', (c) $\sigma_1' \equiv^{cfg} \sigma_2'$, (d) $C_1' = C_2'$, (e) $\mathcal{S}_1' = \mathcal{S}_2'$, (f) $currPc(\sigma_1') \sqsubseteq \ell$ and $currPc(\sigma_2') \sqsubseteq \ell$. There are two cases:

– $(\sigma_1', \sigma_2') = (\sigma_1^0, \sigma_2^0)$. Then, (a) trivially follows since $reach^k(\sigma, \sigma)$ always holds for all $k > 0$ and both $i, j > 0$. Morever, (b)–(f) directly follow from (3)–(7).

– $(\sigma_1', \sigma_2') = (\sigma_1^i, \sigma_2^j)$. Then, (a) directly follows from $\sigma_1^0 \overset{\tau}{\rightsquigarrow}{}^{i} \sigma_1^i$ and $\sigma_2^0 \overset{\tau'}{\rightsquigarrow}{}^{i} \sigma_2^j$. Condition (b) follows from (3), $\langle \Delta_1^i, c_1^i, m_1^i, \langle s_1^i, ctx_1^i \rangle\rangle \approx_\ell \langle \Delta_2^j, c_2^j, m_2^j, \langle s_2^j, ctx_2^j \rangle\rangle$, and $\Delta_1^i(\mathtt{pc}_u) = \Delta_2^j(\mathtt{pc}_u) = \ell'$. Condition (c) follows from $\langle \Delta_1^i, c_1^i, m_1^i, \langle s_1^i, ctx_1^i \rangle\rangle \equiv^{cfg} \langle \Delta_2^j, c_2^j, m_2^j, \langle s_2^j, ctx_2^j \rangle\rangle$. Condition (d) follows from (6), $\mathcal{S}_1^i = \mathcal{S}_1^0$, and $\mathcal{S}_2^j = \mathcal{S}_2^0$. Condition (e) follows from (5) and the fact that we applied only the F-ATOMICSTATEMENT rule, which does not modify the scheduler. Condition (f) follows from (7), $\Delta_1^i(\mathtt{pc}_u) = \Delta_1^0(\mathtt{pc}_u)$, and $\Delta_2^i(\mathtt{pc}_u) = \Delta_2^0(\mathtt{pc}_u)$.

Therefore, the fact that R is a bisimulation directly follows from (i) the fact that (a)–(f) hold for (σ_1^0, σ_2^0) and (σ_1^1, σ_2^1), (ii) assumptions (1), (2), and $\{(\sigma_1^0, \sigma_2^0), (\sigma_1^1, \sigma_2^1)\}$, and (3) there are no configurations that are reachable in i steps from σ_1^0 and j steps from σ_2^0 other than σ_1^i and σ_2^j.

- *first*$(c_1) =$ **while** e **do** c'. The proof of this case is similar to that of *first*$(c_1) =$ **if** e **then** c' **else** c''.

This completes the proof of our claim. □

Finally, Lemma H.17 states a composition result for bisimulations.

Lemma H.17. *Let R_1 be a $(\sigma_0^0, \sigma_1^0, i, j, \ell)$-bisimulation and R_2 be a $(\rho_0^0, \rho_1^0, x, y, \ell')$-bisimulation. If the following conditions hold:*

1) $(\sigma_0^0, \sigma_1^0) \in R_1$,
2) $(\rho_0^0, \rho_1^0) \in R_1 \cap R_2$,
3) $\ell = \ell'$,
4) $\sigma_0^0 \overset{\tau}{\rightsquigarrow}{}^{i} \rho_0^0$, and
5) $\sigma_1^0 \overset{\tau}{\rightsquigarrow}{}^{j} \rho_1^0$,

then $R_1 \cup R_2$ is a $(\sigma_0^0, \sigma_1^0, i+x, j+y, \ell)$-bisimulation.

Proof. Let R_1 be a $(\sigma_0^0, \sigma_1^0, i, j, \ell)$-bisimulation and R_2 be a $(\rho_0^0, \rho_1^0, x, y, \ell')$-bisimulation. Assume, for contradiction's sake, that $R_1 \cup R_2$ is not a $(\sigma_0^0, \sigma_1^0, i+x, j+y, \ell)$-bisimulation. This happens iff there is a $(\nu_0, \nu_1) \in R_1 \cup R_2$ that violates one of the following constraints:

1) $\neg reach^{i+x}(\sigma_0^0, \nu_0)$ or $\neg reach^{j+y}(\sigma_1^0, \nu^1)$. Without loss of generality, we assume that $reach^{i+x}(\sigma_0^0, \nu_0)$ does not hold. If $(\nu_0, \nu_1) \in R_1$, then $reach^i(\sigma_0^0, \nu_0)$ holds and $reach^{i+x}(\sigma_0^0, \nu_0)$ follows, leading to a contradiction. If $(\nu_0,$ $\nu_1) \in R_2 \setminus R_1$, then from $\sigma_0^0 \overset{\tau}{\rightsquigarrow}{}^{i} \rho_0^0$ and $reach^x(\rho_0^0, \nu_0)$, it follows that $reach^{i+x}(\sigma_0^0, \nu_0)$, leading to a contradiction.
2) $\nu_0 \not\approx_\ell \nu_1$. This contradicts $(\nu_0, \nu_1) \in R_1$ or $(\nu_0, \nu_1) \in R_2$.
3) $\nu_0 \not\equiv^{cfg} \nu_1$. This contradicts $(\nu_0, \nu_1) \in R_1$ or $(\nu_0, \nu_1) \in R_2$.
4) $C_0 \neq C_1$, where C_0 is the code in ν_0 and C_1 is the code in ν_1. This contradicts $(\nu_0, \nu_1) \in R_1$ or $(\nu_0, \nu_1) \in R_2$.
5) $\mathcal{S}_0 \neq \mathcal{S}_1$, where $\mathcal{S}_0$ is the scheduler in ν_0 and $\mathcal{S}_1$ is the scheduler in ν_1. This contradicts $(\nu_0, \nu_1) \in R_1$ or $(\nu_0,$ $\nu_1) \in R_2$.
6) $currPc(\nu_0) \not\sqsubseteq \ell$ or $currPc(\nu_1) \not\sqsubseteq \ell$. This contradicts $(\nu_0, \nu_1) \in R_1$ or $(\nu_0, \nu_1) \in R_2$.
7) $\nu_0 \overset{\tau_0}{\rightsquigarrow} \nu_0'$, $\nu_1 \overset{\tau_1}{\rightsquigarrow} \nu_1'$, $reach^{i+x}(\sigma_0^0, \nu_0')$, $reach^{j+y}(\sigma_1^0, \nu_1')$, and $currPc(\nu_0') \sqsubseteq \ell$, but $(\nu_0', \nu_1') \notin R_1 \cup R_2$. There are three cases:
   - $(\nu_0, \nu_1) \in R_0$, $reach^i(\sigma_0^0, \nu_0')$, and $reach^j(\sigma_1^0, \nu_1')$. From this and R_1 is a $(\sigma_0^0, \sigma_1^0, i, j, \ell)$-bisimulation, it follows that $(\nu_0', \nu_1') \in R_1$. Hence, $(\nu_0', \nu_1') \in R_1 \cup R_2$, leading to a contradiction.
   - $(\nu_0, \nu_1) \in R_1$, $reach^x(\rho_0^0, \nu_0')$, and $reach^y(\rho_1^0, \nu_1')$. From this and R_2 is a $(\rho_0^0, \rho_1^0, x, y, \ell)$-bisimulation, it follows that $(\nu_0', \nu_1') \in R_2$. Hence, $(\nu_0', \nu_1') \in R_1 \cup R_2$, leading to a contradiction.

- $(\nu_0, \nu_1) \in R_0$, $\mathit{reach}^{i+x}(\sigma_0^0, \nu_0')$, $\mathit{reach}^{j+y}(\sigma_1^0, \nu_1')$, but $\neg \mathit{reach}^{i}(\sigma_0^0, \nu_0')$ or $\neg \mathit{reach}^{j}(\sigma_1^0, \nu_1')$. This happens iff $(\nu_0, \nu_1) = (\rho_0, \rho_1)$. From this, $(\rho_0, \rho_1) \in R_1 \cap R_2$, $\sigma_0^0 \overset{\tau_0}{\rightsquigarrow}{}^{i} \rho_0^0$, $\sigma_1^0 \overset{\tau_1}{\rightsquigarrow}{}^{j} \rho_1^0$, $\mathit{reach}^{i+x}(\sigma_0^0, \nu_0')$, and $\mathit{reach}^{j+y}(\sigma_1^0, \nu_1')$, it follows that $\mathit{reach}^{x}(\rho_0^0, \nu_0')$ and $\mathit{reach}^{y}(\rho_1^0, \nu_1')$. From this and R_2 is a $(\rho_0^0, \rho_1^0, x, y, \ell)$-bisimulation, it follows that $(\nu_0', \nu_1') \in R_2$. Hence, $(\nu_0', \nu_1') \in R_1 \cup R_2$, leading to a contradiction.

8) $\nu_0 \overset{\tau_0}{\rightsquigarrow} \nu_0'$, $\nu_1 \overset{\tau_1}{\rightsquigarrow} \nu_1'$, $\mathit{reach}^{i+x}(\sigma_0^0, \nu_0')$, $\mathit{reach}^{j+y}(\sigma_1^0, \nu_1')$, $\mathit{currPc}(\nu_0') \not\sqsubseteq \ell$, and there are ν_0'' and ν_1'' such that $\nu_0 \overset{\tau_0'}{\rightsquigarrow}{}^{*} \nu_0''$, $\nu_1 \overset{\tau_1'}{\rightsquigarrow}{}^{*} \nu_1''$, $\mathit{reach}^{i+x}(\sigma_0^0, \nu_0'')$, $\mathit{reach}^{j+y}(\sigma_1^0, \nu_1'')$, $\mathit{currPc}(\nu_0'') \sqsubseteq \ell$, $\mathit{currPc}(\nu_1'') \sqsubseteq \ell$, and $(\nu_0'', \nu_1'') \notin R_1 \cup R_2$. If $\mathit{reach}^{i}(\sigma_0^0, \nu_0'')$ and $\mathit{reach}^{j}(\sigma_1^0, \nu_1'')$, then $(\nu_0'', \nu_1'') \in R_1$ and, therefore, $(\nu_0'', \nu_1'') \in R_1 \cup R_2$, leading to a contradiction. If $\mathit{reach}^{x}(\rho_0^0, \nu_0'')$ and $\mathit{reach}^{y}(\rho_1^0, \nu_1'')$, then $(\nu_0'', \nu_1'') \in R_2$ and, therefore, $(\nu_0'', \nu_1'') \in R_1 \cup R_2$, leading to a contradiction. Note that from $(\rho_0, \rho_1) \in R_1 \cap R_2$, it follows that the above cases are the only possible.

Since all cases lead to a contradiction, this completes the proof of our claim. □

H.9. Proof of the main result

We are now ready to prove the main result of §5, namely that our mechanism provides security with respect to an external attacker atk.

Theorem 4. *For all programs $C = c_1 \cdot \ldots \cdot c_k \in \mathit{Com}_{UID}^{k}$, scheduler $\mathcal{S}$, memories $M = m_1 \cdot \ldots \cdot m_k \in \mathit{Mem}_{UID}^{k}$, and initial runtime state s, whenever $r = \langle \Delta_0, C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \overset{\tau}{\rightsquigarrow}{}^{n} \langle \Delta', C', M', \langle s', \mathit{ctx}'\rangle, \mathcal{S}'\rangle$, then for all $1 \leq i \leq n$, $K_{atk}^{\rightsquigarrow}(\langle M, s\rangle, C, \mathcal{S}, \mathit{trace}(r^{i-1})) \cap A_{atk,sec}(M, s) \subseteq K_{atk}^{\rightsquigarrow}(\langle M, s\rangle, C, \mathcal{S}, \mathit{trace}(r^{i}))$, where the database in r's $(i-1)$-th configuration is $\langle db, U, sec, T, V\rangle$ and $K_{atk}^{\rightsquigarrow}$ refers to Definition 2 with $\rightsquigarrow$ as the underlying evaluation relation.*

Proof. Let $k \in \mathbb{N}$, $C_0 = c_1 \cdot \ldots \cdot c_k \in \mathit{Com}_{UID}^{k}$ be WHILESQL programs, $\mathcal{S}_0$ be a scheduler, $M_0 = m_1 \cdot \ldots \cdot m_k \in \mathit{Mem}_{UID}^{k}$ be memories, s_0 be a database state, σ_0 be the global state $\langle \Delta_0, C_0, M_0, \langle s_0, \epsilon\rangle, \mathcal{S}_0\rangle$. Let σ_1 be a global state and n be a value in $\mathbb{N}$ such that $r = \sigma_0 \overset{\tau}{\rightsquigarrow}{}^{n} \sigma_1$. Assume, for contradiction's sake, that our claim does not hold. Namely, there exists a value $1 \leq i \leq n$ such that $K_{atk}^{\rightsquigarrow}(\langle M_0, s_0\rangle, C_0, \mathcal{S}_0, \mathit{trace}(r^{i-1})) \cap A_{atk,sec}(M_0, s_0) \not\subseteq K_{atk}^{\rightsquigarrow}(\langle M_0, s_0\rangle, C_0, \mathcal{S}, \mathit{trace}(r^{i}))$, where sec is the security policy in the $(i-1)$-th configuration in r. From this, it follows that there is a state $\langle M_1, s_1\rangle$ such that $\langle M_1, s_1\rangle \in K_{atk}^{\rightsquigarrow}(\langle M_0, s_0\rangle, C_0, \mathcal{S}_0, \mathit{trace}(r^{i-1})) \cap A_{atk,sec}(M_0, s_0)$ and $\langle M_1, s_1\rangle \notin K_{atk}^{\rightsquigarrow}(\langle M_0, s_0\rangle, C_0, \mathcal{S}, \mathit{trace}(r^{i}))$. From $\langle M_1, s_1\rangle \in K_{atk}^{\rightsquigarrow}(\langle M_0, s_0\rangle, C_0, \mathcal{S}_0, \mathit{trace}(r^{i-1}))$, it follows that $s_0 \approx_{atk} s_1$, $M_0 \approx_{atk} M_1$, and for all ctx', Δ', τ', C', M', s', $\mathcal{S}'$ such that $\langle \Delta_0, C_0, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}_0\rangle \overset{\tau'}{\rightsquigarrow}{}^{*} \langle \Delta', C', M', \langle s', \mathit{ctx}'\rangle, \mathcal{S}'\rangle$, $\mathit{trace}(r^{i-1}) \sim_{atk} \tau'$. From $\langle M_1, s_1\rangle \in A_{atk,sec}(M, s)$, it follows that $s_1 \approx_{sec,atk} s$ and $M_1 \approx_{atk} M$. Finally, from $\langle M_1, s_1\rangle \notin K_{atk}^{\rightsquigarrow}(\langle M_0, s_0\rangle, C_0, \mathcal{S}, \mathit{trace}(r^{i}))$, it follows that $s_0 \not\approx_{atk} s_1$, $M_0 \not\approx_{atk} M_1$, or there are ctx', Δ', τ', C', M', s', $\mathcal{S}'$ such that $\langle \Delta_0, C_0, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}_0\rangle \overset{\tau'}{\rightsquigarrow}{}^{*} \langle \Delta', C', M', \langle s', \mathit{ctx}'\rangle, \mathcal{S}'\rangle$ and $\mathit{trace}(r^{i}) \not\sim_{atk} \tau'$. Note that only the last case is interesting (since $s_0 \not\approx_{atk} s_1$ and $M_0 \not\approx_{atk} M_1$ immediately contradict $\langle M_1, s_1\rangle \in K_{atk}^{\rightsquigarrow}(\langle M_0, s_0\rangle, C_0, \mathcal{S}, \mathit{trace}(r^{i-1}))$). Therefore, the following conditions hold:

1) $s_1 \approx_{atk} s_0$,
2) $s_1 \approx_{sec,atk} s_0$,
3) $M_1 \approx_{atk} M_0$,
4) there are j, ctx_1^j, Δ_1^j, τ_1^j, C_1^j, M_1^j, s_1^j, $\mathcal{S}_1^j$ such that:
   a) $\langle \Delta_0, C_0, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}_0\rangle \xrightarrow{\tau'}{}^{j} \langle \Delta_1^j, C_1^j, M_1^j, \langle s_1^j, \mathit{ctx}_1^j\rangle, \mathcal{S}_1^j\rangle$,
   b) $\mathit{trace}(r^{i-1}) \sim_{atk} \tau'$, and
   c) $\mathit{trace}(r^{i}) \not\sim_{atk} \tau'$.

In the following, let sec_0 be the policy in the state s_0 (which, from $s_1 \approx_{atk} s_0$, is the same as in s_1), sec be the policy in the $(i-1)$-th configuration in r, and Var_{atk} be the variables occurring in atk's program. Furthermore, let ℓ be the label $\mathit{cl}(\mathit{auth}(sec_0, atk) \cup \mathit{auth}(sec, atk) \cup \bigcup_{x \in \mathit{Var}_{atk}} \mathit{MEM}_x)$ (observe that $\mathit{cl}(\mathit{auth}(sec_0, atk)) \sqsubseteq \ell$ holds), $\sigma_0^{i-1} = \langle \Delta_0^{i-1}, C_0^{i-1}, M_0^{i-1}, \langle s_0^{i-1}, \mathit{ctx}_0^{i-1}\rangle, \mathcal{S}_0^{i-1}\rangle$ be $(i-1)$-th configuration in r, and σ_1^{j-1} be the configuration $\langle \Delta_1^{j-1}, C_1^{j-1}, M_1^{j-1}, \langle s_1^{j-1}, \mathit{ctx}_1^{j-1}\rangle, \mathcal{S}_1^{j-1}\rangle$. From (4.b) and (4.c), it follows that the only interesting cases are those for which $\mathit{trace}(r^{i})\restriction_{atk} = \mathit{trace}(r^{i-1})\restriction_{atk} \cdot \mathit{obs}_0$ (since if $\mathit{trace}(r^{i})\restriction_{atk} = \mathit{trace}(r^{i-1})\restriction_{atk}$, then (4.b) and (4.c) are contradictory statements), where obs_1 is an observation. Therefore, there is a non-empty trace π_1 such that $\mathit{trace}(r^{i-1})\restriction_{atk} = \pi_1$ and $\mathit{trace}(r^{i})\restriction_{atk} = \pi_1 \cdot \mathit{obs}_1$. Let σ_0^{i} and σ_0^{i-1} be the last global states in r^i and r^{i-1}. From $\mathit{trace}(r^{i})\restriction_{atk} = \mathit{trace}(r^{i-1})\restriction_{atk} \cdot \mathit{obs}_0$, it follows that $\sigma_0^{i-1} \overset{\mathit{obs}_0}{\rightsquigarrow} \sigma_0^{i}$. From $\mathit{trace}(r^{i-1}) \sim_{atk} \tau'$, $\mathit{trace}(r^{i}) \not\sim_{atk} \tau'$, and $\mathit{trace}(r^{i})\restriction_{u} = \mathit{trace}(r^{i-1})\restriction_{u} \cdot \mathit{obs}_0$, it follows that $\tau'\restriction_{atk} = \pi_1 \cdot \mathit{obs}_1 \cdot \pi_4$, where obs_1 is an observation different from obs_0 (it this is not the case, this would contradict (4.b) and (4.c) since this would imply $\mathit{trace}(r^{i}) \sim_{atk} \tau'$). Without loss of generality, we assume that $\pi_4 = \epsilon$ and that obs_1 is produced in the last step of $\langle \Delta_0, C_0, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}_0\rangle \xrightarrow{\tau'}{}^{j} \langle \Delta_1^j, C_1^j, M_1^j, \langle s_1^j, \mathit{ctx}_1^j\rangle, \mathcal{S}_1^j\rangle$, i.e., $\sigma_1^{j-1} \overset{\mathit{obs}_1}{\rightsquigarrow} \sigma_1^{j}$. We claim that that (1) $\sigma_0^{i-1} \approx_{\ell} \sigma_1^{j-1}$, (2) $C_0^{i-1} = C_1^{j-1}$, (3) $\mathcal{S}_0^{i-1} = \mathcal{S}_1^{j-1}$, (4) $s_0^{i-1} \equiv^{cfg} s_1^{j-1}$. Furthermore, we also claim that $\mathit{currPc}(\sigma_0^{i-1}) \sqsubseteq \ell$ and $L_{\mathcal{U}}(s_0^{i-1}, atk) \sqsubseteq \ell$. From (1) and $\mathit{currPc}(\sigma_0^{i-1}) \sqsubseteq \ell$, it follows that $\mathit{currPc}(\sigma_1^{j-1}) \sqsubseteq \ell$. From (4) and $L_{\mathcal{U}}(s_0^{i-1}, atk) \sqsubseteq \ell$, it follows that $L_{\mathcal{U}}(s_1^{j-1}, atk) \sqsubseteq \ell$. From this and Lemma H.6, it follows that $\mathit{obs}_0 = \mathit{obs}_1$, leading to a contradiction.

Below, we prove our claims together with other intermediate facts.

Fact 1. We now prove that $\sigma_0^0 \approx_{\ell} \sigma_1^0$, where $\sigma_0^0 = \langle \Delta_0, C_0, M_0, \langle s_0, \epsilon\rangle, \mathcal{S}_0\rangle$, $\sigma_1^0 = \langle \Delta_0, C_0, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}_0\rangle$, $s_0 = \langle db_0, U_0, sec_0, T_0, V_0\rangle$, and $s_1 = \langle db_1, U_1, sec_1, T_1, V_1\rangle$. To do so, we need to show that:

- For all queries $q \in RC$ such that $L_{\mathcal{Q}}(\Delta_0, q) \sqsubseteq \ell$, $[q]^{db_0} = [q]^{db_1}$. From $\ell = cl(auth(sec_0, atk) \cup auth(sec, atk) \cup \bigcup_{x \in Var_{atk}} MEM_x)$ and q is a query (i.e., it does not refer to MEM_x for any x), it follows that $L_{\mathcal{Q}}(\Delta_0, q) \sqsubseteq cl(auth(sec_0, atk) \cup auth(sec, atk))$. From $L_{\mathcal{Q}}$'s definition, it follows that $L_{\mathcal{Q}}(\Delta_0, q) = \bigsqcup_{Q \in supp_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} \Delta_0(q')$. From this and Δ_0's definition, it follows that $\Delta_0(q) = cl(q)$ for all $q \in RC^{pred}$. From this and $L_{\mathcal{Q}}(\Delta_0, q) = \bigsqcup_{Q \in supp_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} \Delta_0(q')$, it follows that $L_{\mathcal{Q}}(\Delta_0, q) = \bigsqcup_{Q \in supp_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} cl(q')$. From this and $L_{\mathcal{Q}}(\Delta_0, q) \sqsubseteq cl(auth(sec_0, atk) \cup auth(sec, atk))$, it follows $\bigsqcup_{Q \in supp_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} cl(q') \sqsubseteq cl(auth(sec_0, atk) \cup auth(sec, atk))$. Furthermore, from the definition of $supp_{D,\Gamma}(q)$, it follows that $cl(q) \sqsubseteq \bigsqcup_{Q \in supp_{D,\Gamma}(q)} \bigsqcup_{q' \in Q} cl(q')$. Therefore, the tables and views in $auth(sec_0, atk) \cup auth(sec, atk)$ determine the values of the queries in $supp_{D,\Gamma}(q)$, which in turn determine the value of q, i.e., $cl(q) \sqsubseteq cl(auth(sec_0, atk) \cup auth(sec, atk))$. From $s_1 \approx_{atk} s_0$, it follows that the content of all tables and views in $auth(sec_0, atk)$ is the same in s_0 and s_1. From $s_1 \approx_{sec,atk} s_0$, it follows that the content of all tables and views in $auth(sec, atk)$ is the same in s_0 and s_1. From this and $cl(q) \sqsubseteq cl(auth(sec_0, atk) \cup auth(sec, atk))$, it follows that $[q]^{db_0} = [q]^{db_1}$.
- For all variables $x \in Var$ such that $\Delta_0(x) \sqsubseteq \ell$, $[\![M_0]\!](x) = [\![M_1]\!](x)$. From $\ell = cl(auth(sec_0, atk) \cup auth(sec, atk) \cup \bigcup_{x \in Var_{atk}} MEM_x)$, it follows that $\Delta_0(x) \sqsubseteq cl(auth(sec_0, atk) \cup auth(sec, atk) \cup \bigcup_{x \in Var_{atk}} MEM_x)$. From this and $\Delta_0(x) = \top$ if $x \notin Var_{atk}$ and $\Delta_0(x) = MEM_x$ otherwise, it follows that $\Delta_0(x) = MEM_x$. From this, it follows that $x \in Var_{atk}$. From this and $s_1 \approx_{atk} s_0$, it follows that $[\![M_0]\!](x) = [\![M_1]\!](x)$.

These facts together with the fact that the monitor state is the same in σ_0^0 and σ_1^0, leads to $\sigma_0^0 \approx_\ell \sigma_1^0$.

Fact 2. We now prove that $s_0 \equiv^{cfg} s_1$. Let $s_0 = \langle db_0, U_0, S_0, T_0, V_0 \rangle$ and $s_1 = \langle db_1, U_1, S_1, T_1, V_1 \rangle$. From $s_1 \approx_{atk} s_0$ and $\approx_{atk}$'s definition, it follows that $U_0 = U_1$, $S_0 = S_1$, $T_0 = T_1$, and $V_0 = V_1$. From this and $\equiv^{cfg}$'s definition, it follows that $s_0 \equiv^{cfg} s_1$.

Fact 3. We now show that $currPc(\sigma_0^{i-1}) \sqsubseteq \ell$ and $L_{\mathcal{U}}(s_0^{i-1}, atk) \sqsubseteq \ell$. From $\ell = cl(auth(sec_0, atk) \cup auth(sec, atk) \cup \bigcup_{x \in Var_{atk}} MEM_x)$ and sec is the policy in σ_0^{i-1}, it immediately follows that $L_{\mathcal{U}}(s_0^{i-1}, atk) \sqsubseteq \ell$. Assume, for contradiction's sake, that $currPc(\sigma_0^{i-1}) \not\sqsubseteq \ell$. From $\sigma_0^{i-1} \overset{obs_0}{\rightsquigarrow} \sigma_0^i$ and $obs_0 \neq \epsilon$, it follows that the executed rule produced an event. In the following, let $currUsr(\sigma_0^{i-1}) = u$. Observe that $\Delta_0^{i-1}(\mathtt{pc}_u) = currPc(\sigma_0^{i-1})$. There are two cases:

- Rule F-EVAL-STEP. From the rule, it follows that $\langle \Delta_0^{i-1}, c_0^{i-1}, m_0^{i-1}, \langle s_0^{i-1}, ctx_0^{i-1} \rangle \rangle \overset{obs_0}{\rightsquigarrow}_u \langle \Delta_0^i, c_0^i, m_0^i, \langle s_0^i, ctx_0^i \rangle \rangle$ (which we denote r). From this and Lemma H.1, it follows that $\Delta_0^{i-1}(deps(obs_0, r)) \sqcup \Delta_0^{i-1}(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s_0^{i-1}, user(obs_0))$. From this, it follows that $\Delta_0^{i-1}(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s_0^{i-1}, user(obs_0))$. Furthermore, since $trace(r^i)\upharpoonright_{atk} = trace(r^{i-1})\upharpoonright_{atk} \cdot obs_0$, it follows that $user(obs_0) = atk$. From this and $\Delta_0^{i-1}(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s_0^{i-1}, user(obs_0))$, it follows that $\Delta_0^{i-1}(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(s_0^{i-1}, atk)$. From this and $L_{\mathcal{U}}(s_0^{i-1}, atk) \sqsubseteq \ell$, it follows that $\Delta_0^{i-1}(\mathtt{pc}_u) \sqsubseteq \ell$, leading to a contradiction.
- Rule F-ATOMIC-STATEMENT. The proof is similar to that of F-EVAL-STEP.

Since all cases lead to a contradiction, this completes the proof of Fact 3.

Fact 4. We prove that $s_0^{i-1} \equiv^{cfg} s_1^{j-1}$. From $\sigma_0^0 \overset{\tau_0}{\rightsquigarrow}^* \sigma_0^{i-1} \overset{obs_0}{\rightsquigarrow} \sigma_0^i$, $\sigma_1^0 \overset{\tau_1}{\rightsquigarrow}^* \sigma_1^{j-1} \overset{obs_1}{\rightsquigarrow} \sigma_1^j$, $\tau_0\upharpoonright_{atk} = \tau_1\upharpoonright_{atk}$, all configuration changes are associated with public events, and the code produced by the expansion process either terminates or gets stuck, it follows that the configuration has been modified in the same way in $\sigma_0^0 \overset{\tau_0}{\rightsquigarrow}^* \sigma_0^{i-1}$ and $\sigma_1^0 \overset{\tau_1}{\rightsquigarrow}^* \sigma_1^{j-1}$. Therefore, $s_0^{i-1} \equiv^{cfg} s_1^{j-1}$.

Fact 5. We now prove that $currPc(\sigma_1^{j-1}) \sqsubseteq \ell$. From $\sigma_1^{j-1} \overset{obs_1}{\rightsquigarrow} \sigma_1^j$, there are two cases:

1) We applied the F-OUT rule. From the rule, it follows that $currPc(\sigma_1^{j-1}) \sqsubseteq L_{\mathcal{U}}(s_1^{j-1}, u')$. Furthermore, since obs_1 is visible to atk, it follows that $u' = atk$. Hence, $currPc(\sigma_1^{j-1}) \sqsubseteq L_{\mathcal{U}}(s_1^{j-1}, atk)$. From this and $s_0^{i-1} \equiv^{cfg} s_1^{j-1}$ (Fact 4), it follows that $currPc(\sigma_1^{j-1}) \sqsubseteq L_{\mathcal{U}}(s_0^{i-1}, atk)$. From this and $L_{\mathcal{U}}(s_0^{i-1}, atk) \sqsubseteq \ell$ (Fact 3), it follows that $currPc(\sigma_1^{j-1}) \sqsubseteq \ell$.
2) We applied the F-UPDATECONFIGURATIONOK. From the rule, it follows that $currPc(\sigma_1^{j-1}) \sqsubseteq auth(sec_0, atk) \sqsubseteq L_{\mathcal{U}}(s_1^{j-1}, atk)$. From this and $s_0^{i-1} \equiv^{cfg} s_1^{j-1}$ (Fact 4), it follows that $currPc(\sigma_1^{j-1}) \sqsubseteq L_{\mathcal{U}}(s_0^{i-1}, atk)$. From this and $L_{\mathcal{U}}(s_0^{i-1}, atk) \sqsubseteq \ell$ (Fact 3), it follows that $currPc(\sigma_1^{j-1}) \sqsubseteq \ell$.

Fact 6. We now prove that (1) $\sigma_0^{i-1} \approx_\ell \sigma_1^{j-1}$, (2) $C_0^{i-1} = C_1^{j-1}$, and (3) $\mathcal{S}_0^{i-1} = \mathcal{S}_1^{j-1}$. From Facts 1 and 2, it follows that initially $\sigma_0^0 \approx_\ell \sigma_1^0$, $C_0^0 = C_1^0$, $\mathcal{S}_0^0 = \mathcal{S}_1^0$, and $s_0^0 \equiv^{cfg} s_1^0$. Furthermore, $currPc(\sigma_0^0) = currPc(\sigma_1^0) = \bot$ given that both runs start from the initial monitor state Δ_0. Therefore, we can repeatedly apply Lemmas H.15 and H.16 (depending on whether we are in a low or high context given ℓ) to construct bisimulations among states in the two runs and use Lemma H.17 to compose the various bisimulations in a unique bisimulation R. We remark that during the construction of the bisimulation we can always apply either Lemma H.15 or Lemma H.16. In particular, $currPc(\sigma_0^{i-1}) \sqsubseteq \ell$ (Fact 3) and $currPc(\sigma_1^{j-1}) \sqsubseteq \ell$ (Fact 5) ensure that the execution of branching statements leading to high contexts with respect to ℓ always terminates before σ_0^{j-1} and σ_1^{j-1}. Finally, observe that $\{(\sigma_0^0, \sigma_0^1), (\sigma_0^{i-1}, \sigma_1^{j-1})\} \subseteq R$ by construction. From this, $\sigma_0^{i-1} \approx_\ell \sigma_1^{j-1}$, $C_0^{i-1} = C_1^{j-1}$, and $\mathcal{S}_0^{i-1} = \mathcal{S}_1^{j-1}$ directly follow. $\square$

Appendix I. Approximations for disclosure lattices

Here, we provide all the proofs about approximations of disclosure lattices (see §6).

I.1. Soundness criteria

Equipping our monitor with arbitrary approximations may break the monitor's security guarantees. To avoid that, we introduce sound approximations, and we prove that they preserve the monitor's guarantees. We say that an approximation $\langle \mathcal{L}^{abs}, \sqsubseteq^{abs}, \sqcup^{abs}, \Delta_0^{abs}, L_{\mathcal{Q}}^{abs}, L_{\mathcal{U}}^{abs}, \mathit{auth}^{abs}, \gamma^-, \gamma^+\rangle$ is *sound* if:

1) The lower and upper bounds functions are well-defined. Namely, we require that for each label $\ell \in \mathcal{L}^{abs}$, $\gamma^-(\ell) \sqsubseteq \gamma^+(\ell)$.
2) The abstract ordering relation $\sqsubseteq^{abs}$ approximates $\sqsubseteq$. Formally, for any two abstract labels $\ell_1, \ell_2 \in \mathcal{L}^{abs}$, we require that whenever $\ell_1 \sqsubseteq^{abs} \ell_2$ holds, then $\gamma^+(\ell_1) \sqsubseteq \gamma^-(\ell_2)$.
   By using the upper bound γ^+ on the left-hand side of $\sqsubseteq$ and the lower bound γ^- on the right-hand side, we ensure that whenever the monitor (equipped with the approximation) determines that a check is satisfied, then the check is satisfied as well with respect to the disclosure lattice.
3) The abstract join operator $\sqcup^{abs}$ approximates $\sqcup$. Formally, we require that, for any two abstract labels $\ell_1, \ell_2 \in \mathcal{L}^{abs}$, $\gamma^-(\ell_1 \sqcup^{abs} \ell_2) \sqsubseteq \gamma^-(\ell_1) \sqcup \gamma^-(\sqcup_2)$ and $\gamma^+(\ell_1) \sqcup \gamma^+(\ell_2) \sqsubseteq \gamma^+(\ell_1 \sqcup^{abs} \ell_2)$. This ensures that joining abstract labels does not lose information with respect to the disclosure lattice.
4) The initial abstract monitor state Δ_0^{abs} approximates Δ_0. Formally, we require $\Delta_0^{abs} \sqsubseteq^-_{Var \cup RC^{pred} \cup \{\mathtt{pc}_u \mid u \in UID\}} \Delta_0 \sqsubseteq^+_{Var \cup RC^{pred} \cup \{\mathtt{pc}_u \mid u \in UID\}} \Delta_0^{abs}$, where $\Delta^{abs} \sqsubseteq^-_V \Delta$ denotes that $\gamma^-(\Delta^{abs}(x)) \sqsubseteq \Delta(x)$ and $\Delta \sqsubseteq^+_V \Delta^{abs}$ denotes that $\Delta(x) \sqsubseteq \gamma^+(\Delta^{abs}(x))$ for all $x \in V$. This ensures that the abstract initial state Δ_0^{abs} contains at least as much information as Δ_0 about all variables and tuples.
5) The abstract mapping $L_{\mathcal{Q}}^{abs}$ approximates $L_{\mathcal{Q}}$. More precisely, we require that $L_{\mathcal{Q}}^{abs}(\Delta^{abs}, q)$ approximates $L_{\mathcal{Q}}(\Delta, q)$ whenever Δ^{abs} approximates Δ for all predicate queries in q's support. Formally, whenever $\Delta^{abs} \sqsubseteq^-_{supp(q)} \Delta \sqsubseteq^+_{supp(q)} \Delta^{abs}$, then $\gamma^-(L_{\mathcal{Q}}^{abs}(\Delta^{abs}, q)) \sqsubseteq L_{\mathcal{Q}}(\Delta, q) \sqsubseteq \gamma^+(L_{\mathcal{Q}}^{abs}(\Delta^{abs}, q))$ must hold. This guarantees that $L_{\mathcal{Q}}^{abs}(\Delta^{abs}, q)$ always captures at least as much information as $L_{\mathcal{Q}}(\Delta, q)$. Moreover, we also require that $L_{\mathcal{Q}}^{abs}(\Delta^{abs}, q) = \Delta^{abs}(q)$ for any abstract state Δ^{abs} and predicate query $q \in RC^{pred}$.
6) The abstract mapping $L_{\mathcal{U}}^{abs}$ approximates $L_{\mathcal{U}}$. That is, for any system state $s \in \Omega_M$ and user $u \in UID$, we require that $\gamma^-(L_{\mathcal{U}}^{abs}(s, u)) \sqsubseteq L_{\mathcal{U}}(s, u) \sqsubseteq \gamma^+(L_{\mathcal{U}}^{abs}(s, u))$. Since $L_{\mathcal{U}}(s, u)$ maps u to the element in $\mathcal{L}$ that captures u's privileges given the current policy, this requirement in particular ensures that $\gamma^-(L_{\mathcal{U}}^{abs}(s, u))$ does not grant more privileges to u than specified in the policy.
7) The abstract mapping auth^{abs} approximates auth. Formally, for any security policy sec and user $u \in UID$, we require $\gamma^-(\mathit{auth}^{abs}(\mathit{sec}, u)) \sqsubseteq \mathit{auth}(\mathit{sec}, u) \sqsubseteq \gamma^+(\mathit{auth}^{abs}(\mathit{sec}, u))$.

I.2. Using approximations

Let $\mathcal{A}$ be an approximation $\langle \mathcal{L}^{abs}, \sqsubseteq^{abs}, \sqcup^{abs}, \Delta_0^{abs}, L_{\mathcal{Q}}^{abs}, L_{\mathcal{U}}^{abs}, \mathit{auth}^{abs}, \gamma^-, \gamma^+\rangle$. In the following, we use the phrase "equipping our monitor with $\mathcal{A}$" to denote the evaluation relation obtained by replacing $\sqsubseteq$ with $\sqsubseteq^{abs}$, $\sqcup$ with $\sqcup^{abs}$, Δ_0 with Δ_0^{abs}, $L_{\mathcal{Q}}$ with $L_{\mathcal{Q}}^{abs}$, $L_{\mathcal{U}}$ with $L_{\mathcal{U}}^{abs}$, and $cl(\mathit{auth}(\mathit{sec}_0, \mathit{atk}))$ with $\mathit{auth}^{abs}(\mathit{sec}_0, \mathit{atk})$.

I.3. Sound approximations preserve security

Proposition I.1 states that equipping our monitor with a sound approximation does not introduce new behaviors in the monitor's semantics.

Proposition I.1. *Let $V = Var \cup RC^{pred} \cup \{\mathtt{pc}_u \mid u \in UID\}$ and $\mathcal{A}$ be a sound approximation and $\rightsquigarrow_{\mathcal{A}}$ be the evaluation relation obtained by equipping our monitor with $\mathcal{A}$. Whenever there is a run $r = \langle \Delta_0^{abs}, C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \overset{\tau}{\rightsquigarrow}{}^n_{\mathcal{A}} \langle \Delta^{abs\prime}, C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$, then there is a concrete monitor state Δ' and a sequence of programs C'' such that (1) $\langle \Delta_0, C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \overset{\tau}{\rightsquigarrow}{}^n \langle \Delta', C'', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$, (2) $\Delta^{abs\prime} \sqsubseteq^-_V \Delta' \sqsubseteq^+_V \Delta^{abs\prime}$, and (3) C and C' are identical except for statements of the form* **set pc to** *ℓ and for each* **set pc to** *ℓ statement in C' the corresponding* **set pc to** *ℓ' statement in C'' is such that $\gamma^-(\ell) \sqsubseteq \ell' \sqsubseteq \gamma^+(\ell)$.*

Proof. We prove our claim by induction on the length n of the run r.

For the base case, let $n = 0$. Then, $r = \langle \Delta_0^{abs}, C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \rightsquigarrow^0_{\mathcal{A}} \langle \Delta_0^{abs}, C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle$. The corresponding run is $\langle \Delta_0, C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \rightsquigarrow^0 \langle \Delta_0, C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle$ and $\Delta_0^{abs} \sqsubseteq^-_V \Delta_0 \sqsubseteq^+_V \Delta_0^{abs}$ directly follows from $\mathcal{A}$'s soundness (requirement 4).

For the induction's step, we assume that the claim holds for all runs of length $n-1$ and we show that it holds also for $r = \langle \Delta_0^{abs}, C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \overset{\tau}{\rightsquigarrow}{}^n_{\mathcal{A}} \langle \Delta^{abs\prime}, C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$. From the induction hypothesis, we know that given $\langle \Delta_0^{abs}, C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \overset{\mathcal{I}}{\rightsquigarrow}_{\mathcal{A}}^{n-1} \langle \Delta^{abs''}, C'', M'', \langle s'', ctx''\rangle, \mathcal{S}''\rangle$, there is a corresponding run $\langle \Delta_0, C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \overset{\mathcal{I}}{\rightsquigarrow}^{n-1} \langle \Delta'', C_1'', M'', \langle s'', ctx''\rangle, \mathcal{S}''\rangle$ such that (a) $\Delta^{abs''} \sqsubseteq_V^- \Delta'' \sqsubseteq_V^+ \Delta^{abs''}$, and (b) for each **set pc to** ℓ statement in C'' the corresponding **set pc to** ℓ' statement in C_1'' is such that $\gamma^-(\ell) \sqsubseteq \ell' \sqsubseteq \gamma^+(\ell)$. We assume that we can make another step of the execution according to $\rightsquigarrow_{\mathcal{A}}$, that is, $\langle \Delta^{abs''}, C'', M'', \langle s'', ctx''\rangle, \mathcal{S}''\rangle \overset{\mathcal{I}}{\rightsquigarrow}_{\mathcal{A}} \langle \Delta^{abs'}, C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$. We now prove our claim by case distinction depending on the applied rule in the local semantics (with a slight abuse of notation, we ignore the global semantics):

- Rule F-ASSIGN. Since we successfully applied the rule according to $\rightsquigarrow_{\mathcal{A}}$, it follows that (i) $\Delta^{abs''}(\mathtt{pc}_u) \sqsubseteq^{abs} auth^{abs}(sec_0, atk) \vee \Delta^{abs''}(\mathtt{pc}_u) \sqsubseteq^{abs} \Delta^{abs''}(x)$ and (ii) $\Delta^{abs'} = \Delta^{abs''}[x \mapsto \Delta^{abs''}(\mathtt{pc}_u) \sqcup^{abs} \Delta^{abs''}(e)]$. From (i), there are two cases:
  - Assume $\Delta^{abs''}(\mathtt{pc}_u) \sqsubseteq^{abs} auth^{abs}(sec_0, atk)$ holds. From $\mathcal{A}$'s soundness (requirement 2), it follows that $\gamma^+(\Delta^{abs''}(\mathtt{pc}_u)) \sqsubseteq \gamma^-(auth^{abs}(sec_0, atk))$. From $\Delta'' \sqsubseteq_V^+ \Delta^{abs''}$ (see (a)), it follows that $\Delta''(\mathtt{pc}_u) \sqsubseteq \gamma^+(\Delta^{abs''}(\mathtt{pc}_u))$. From $\mathcal{A}$'s soundness (requirement 7), it follows that $\gamma^-(auth^{abs}(sec_0, atk)) \sqsubseteq cl(auth(sec_0, atk))$. From $\gamma^+(\Delta^{abs''}(\mathtt{pc}_u)) \sqsubseteq \gamma^-(auth^{abs}(sec_0, atk))$, $\Delta''(\mathtt{pc}_u) \sqsubseteq \gamma^+(\Delta^{abs''}(\mathtt{pc}_u))$, and $\gamma^-(auth^{abs}(sec_0, atk)) \sqsubseteq cl(auth(sec_0, atk))$, we therefore have $\Delta''(\mathtt{pc}_u) \sqsubseteq cl(auth(sec_0, atk))$. Hence, $\Delta''(\mathtt{pc}_u) \sqsubseteq cl(auth(sec_0, atk)) \vee \Delta''(\mathtt{pc}_u) \sqsubseteq \Delta''(x)$ holds.
  - Assume that $\Delta^{abs''}(\mathtt{pc}_u) \sqsubseteq^{abs} \Delta^{abs''}(x)$ holds. From $\mathcal{A}$'s soundness (requirement 2), it follows that $\gamma^+(\Delta^{abs''}(\mathtt{pc}_u)) \sqsubseteq \gamma^-(\Delta^{abs''}(x))$. From $\Delta'' \sqsubseteq_V^+ \Delta^{abs''}$ (see (a)), it follows that $\Delta''(\mathtt{pc}_u) \sqsubseteq \gamma^+(\Delta^{abs''}(\mathtt{pc}_u))$. From $\Delta^{abs''} \sqsubseteq_V^- \Delta''$ (see (a)), it follows that $\gamma^-(\Delta^{abs''}(x)) \sqsubseteq \Delta''(x)$. From $\gamma^+(\Delta^{abs''}(\mathtt{pc}_u)) \sqsubseteq \gamma^-(\Delta^{abs''}(x))$, $\Delta''(\mathtt{pc}_u) \sqsubseteq \gamma^+(\Delta^{abs''}(\mathtt{pc}_u))$, and $\gamma^-(\Delta^{abs''}(x)) \sqsubseteq \Delta''(x)$, we have $\Delta''(\mathtt{pc}_u) \sqsubseteq \Delta''(x)$. Hence, $\Delta''(\mathtt{pc}_u) \sqsubseteq cl(auth(sec_0, atk)) \vee \Delta''(\mathtt{pc}_u) \sqsubseteq \Delta''(x)$ holds.

  Since $\Delta''(\mathtt{pc}_u) \sqsubseteq cl(auth(sec_0, atk)) \vee \Delta''(\mathtt{pc}_u) \sqsubseteq \Delta''(x)$ holds in both cases, we can apply the rule F-ASSIGN even according to $\rightsquigarrow$. Hence, we have $\langle \Delta'', C_1'', M'', \langle s'', ctx''\rangle, \mathcal{S}''\rangle \rightsquigarrow \langle \Delta', C_1', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$, where $\Delta' = \Delta''[x \mapsto \Delta''(\mathtt{pc}_u) \sqcup \Delta''(e)]$. We still have to show that $\Delta^{abs'} \sqsubseteq_V^- \Delta' \sqsubseteq_V^+ \Delta^{abs'}$ and that the labels occurring in C_1' are consistent with those in C'. The requirement on C_1' follows directly from the induction hypothesis (see (b)). Showing $\Delta^{abs'} \sqsubseteq_V^- \Delta' \sqsubseteq_V^+ \Delta^{abs'}$ is equivalent to showing that $\gamma^-(\Delta^{abs'}(x)) \sqsubseteq \Delta'(x) \sqsubseteq \gamma^+(\Delta^{abs'}(x))$. We remark that $\Delta^{abs'}(x) = \Delta^{abs''}(\mathtt{pc}_u) \sqcup^{abs} \Delta^{abs''}(e)$ (from (2)) and $\Delta'(x) = \Delta''(\mathtt{pc}_u) \sqcup \Delta''(e)$ (from the F-ASSIGN rule). We first show that $\gamma^-(\Delta^{abs'}(x)) \sqsubseteq \Delta'(x)$. From $\mathcal{A}$'s soundness (requirement 3), $\gamma^-(\Delta^{abs''}(\mathtt{pc}_u) \sqcup^{abs} \Delta^{abs''}(e)) \sqsubseteq \gamma^-(\Delta^{abs''}(\mathtt{pc}_u)) \sqcup \gamma^-(\Delta^{abs''}(e))$. From $\Delta^{abs''} \sqsubseteq_V^- \Delta''$ (see (a)), it follows that $\gamma^-(\Delta^{abs''}(\mathtt{pc}_u)) \sqsubseteq \Delta''(\mathtt{pc}_u)$ and $\gamma^-(\Delta^{abs''}(e)) \sqsubseteq \Delta''(e)$, and, therefore, $\gamma^-(\Delta^{abs''}(\mathtt{pc}_u)) \sqcup \gamma^-(\Delta^{abs''}(e)) \sqsubseteq \Delta''(\mathtt{pc}_u) \sqcup \Delta''(e)$. From this and $\gamma^-(\Delta^{abs''}(\mathtt{pc}_u) \sqcup^{abs} \Delta^{abs''}(e)) \sqsubseteq \gamma^-(\Delta^{abs''}(\mathtt{pc}_u)) \sqcup \gamma^-(\Delta^{abs''}(e))$, we have $\gamma^-(\Delta^{abs''}(\mathtt{pc}_u) \sqcup^{abs} \Delta^{abs''}(e)) \sqsubseteq \Delta''(\mathtt{pc}_u) \sqcup \Delta''(e)$ or, equivalently, $\gamma^-(\Delta^{abs'}(x)) \sqsubseteq \Delta'(x)$.

  We now show that $\Delta'(x) \sqsubseteq \gamma^+(\Delta^{abs'}(x))$. From $\mathcal{A}$'s soundness (requirement 3), $\gamma^+(\Delta^{abs''}(\mathtt{pc}_u)) \sqcup \gamma^+(\Delta^{abs''}(e)) \sqsubseteq \gamma^+(\Delta^{abs''}(\mathtt{pc}_u) \sqcup^{abs} \Delta^{abs''}(e))$. From $\Delta'' \sqsubseteq_V^+ \Delta^{abs''}$ (see (a)), it follows that $\Delta''(\mathtt{pc}_u) \sqsubseteq \gamma^+(\Delta^{abs''}(\mathtt{pc}_u))$ and $\Delta''(e) \sqsubseteq \gamma^+(\Delta^{abs''}(e))$. Hence, $\Delta''(\mathtt{pc}_u) \sqcup \Delta''(e) \sqsubseteq \gamma^+(\Delta^{abs''}(\mathtt{pc}_u)) \sqcup \gamma^+(\Delta^{abs''}(e))$. From this and $\gamma^+(\Delta^{abs''}(\mathtt{pc}_u)) \sqcup \gamma^+(\Delta^{abs''}(e)) \sqsubseteq \gamma^+(\Delta^{abs''}(\mathtt{pc}_u) \sqcup^{abs} \Delta^{abs''}(e))$, we have $\Delta''(\mathtt{pc}_u) \sqcup \Delta''(e) \sqsubseteq \gamma^+(\Delta^{abs''}(\mathtt{pc}_u) \sqcup^{abs} \Delta^{abs''}(e))$ or, equivalently, $\Delta'(x) \sqsubseteq \gamma^+(\Delta^{abs'}(x))$.
- Rule F-OUT. Since we successfully applied the rule according to $\rightsquigarrow_{\mathcal{A}}$, it follows that $\Delta^{abs''}(e) \sqcup^{abs} \Delta^{abs''}(\mathtt{pc}_u) \sqsubseteq^{abs} L_{\mathcal{U}}^{abs}(\langle s'', ctx''\rangle, u')$ and $\tau = \langle u', [\![e]\!](m'')\rangle$ (where m'' is the current memory in M''). From $\Delta^{abs''}(e) \sqcup^{abs} \Delta^{abs''}(\mathtt{pc}_u) \sqsubseteq^{abs} L_{\mathcal{U}}^{abs}(\langle s'', ctx''\rangle, u')$ and $\mathcal{A}$'s soundness (requirement 2), it follows that $\gamma^+(\Delta^{abs''}(e) \sqcup^{abs} \Delta^{abs''}(\mathtt{pc}_u)) \sqsubseteq \gamma^-(L_{\mathcal{U}}^{abs}(\langle s'', ctx''\rangle, u'))$. From $\mathcal{A}$'s soundness (requirement 6), it follows that $\gamma^-(L_{\mathcal{U}}^{abs}(\langle s'', ctx''\rangle, u')) \sqsubseteq L_{\mathcal{U}}(\langle s'', ctx''\rangle, u')$. From $\mathcal{A}$'s soundness (requirement 3), it follows that $\gamma^+(\Delta^{abs''}(e)) \sqcup \gamma^+(\Delta^{abs''}(\mathtt{pc}_u))) \sqsubseteq \gamma^+(\Delta^{abs''}(e) \sqcup^{abs} \Delta^{abs''}(\mathtt{pc}_u))$. From $\Delta'' \sqsubseteq_V^+ \Delta^{abs''}$ (see (a)), it follows that $\Delta''(\mathtt{pc}_u) \sqsubseteq \gamma^+(\Delta^{abs''}(\mathtt{pc}_u))$ and $\Delta''(e) \sqsubseteq \gamma^+(\Delta^{abs''}(e))$ and, therefore, $\Delta''(e) \sqcup \Delta''(\mathtt{pc}_u) \sqsubseteq \gamma^+(\Delta^{abs''}(e)) \sqcup \gamma^+(\Delta^{abs''}(\mathtt{pc}_u)))$. Hence, $\Delta''(e) \sqcup \Delta''(\mathtt{pc}_u) \sqsubseteq L_{\mathcal{U}}(\langle s'', ctx''\rangle, u')$. Therefore, we can apply the rule F-OUT even according to $\rightsquigarrow$. Hence, we have $\langle \Delta'', C_1'', M'', \langle s'', ctx''\rangle, \mathcal{S}''\rangle \overset{\mathcal{I}}{\rightsquigarrow} \langle \Delta', C_1', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$, where $\Delta' = \Delta''$ and $\tau = \langle u', [\![e]\!](m'')\rangle$. Observe that $\Delta^{abs'} \sqsubseteq_V^- \Delta' \sqsubseteq_V^+ \Delta^{abs'}$ directly follows from $\Delta^{abs''} \sqsubseteq_V^- \Delta'' \sqsubseteq_V^+ \Delta^{abs''}$ (see (a)), $\Delta' = \Delta''$, and $\Delta^{abs'} = \Delta^{abs''}$. Moreover, the requirement on C_1' and C' directly follows from the induction hypothesis (see (b)).
- Rule F-IFTRUE. We can immediately apply the rule F-IFTRUE also according to $\rightsquigarrow$ and we get $\langle \Delta'', C_1'', M'', \langle s'', ctx''\rangle, \mathcal{S}''\rangle \overset{\mathcal{I}}{\rightsquigarrow} \langle \Delta', C_1', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$. Then, $\Delta' = \Delta''[\mathtt{pc}_u \mapsto \Delta''(e) \sqcup \Delta''(\mathtt{pc}_u)]$ and $\Delta^{abs'} = \Delta^{abs''}[\mathtt{pc}_u \mapsto \Delta^{abs''}(e) \sqcup^{abs} \Delta^{abs''}(\mathtt{pc}_u)]$. From $\Delta^{abs''} \sqsubseteq_V^- \Delta'' \sqsubseteq_V^+ \Delta^{abs''}$ (see (a)), it follows that $\gamma^-(\Delta^{abs''}(e)) \sqsubseteq \Delta''(e) \sqsubseteq \gamma^+(\Delta^{abs''}(e))$ and $\gamma^-(\Delta^{abs''}(\mathtt{pc}_u)) \sqsubseteq \Delta''(\mathtt{pc}_u) \sqsubseteq \gamma^+(\Delta^{abs''}(\mathtt{pc}_u))$. From $\mathcal{A}$'s soundness (requirement 3), we have that $\gamma^-(\Delta^{abs''}(e) \sqcup^{abs} \Delta^{abs''}(\mathtt{pc}_u)) \sqsubseteq \gamma^-(\Delta^{abs''}(e)) \sqcup \gamma^-(\Delta^{abs''}(\mathtt{pc}_u))$ and $\gamma^+(\Delta^{abs''}(e)) \sqcup \gamma^+(\Delta^{abs''}(\mathtt{pc}_u)) \sqsubseteq \gamma^+(\Delta^{abs''}(e) \sqcup^{abs} \Delta^{abs''}(\mathtt{pc}_u))$. From $\gamma^-(\Delta^{abs''}(e) \sqcup^{abs} \Delta^{abs''}(\mathtt{pc}_u)) \sqsubseteq \gamma^-(\Delta^{abs''}(e)) \sqcup \gamma^-(\Delta^{abs''}(\mathtt{pc}_u))$, $\gamma^-(\Delta^{abs''}(e)) \sqsubseteq \Delta''(e)$, and $\gamma^-(\Delta^{abs''}(\mathtt{pc}_u)) \sqsubseteq \Delta''(\mathtt{pc}_u)$, it follows $\gamma^-(\Delta^{abs''}(e) \sqcup^{abs} \Delta^{abs''}(\mathtt{pc}_u)) \sqsubseteq \Delta''(e) \sqcup \Delta''(\mathtt{pc}_u)$ or, equivalently, $\gamma^-(\Delta^{abs'}(\mathtt{pc}_u)) \sqsubseteq \Delta^{abs'}(\mathtt{pc}_u)$. From $\gamma^+(\Delta^{abs''}(e)) \sqcup \gamma^+(\Delta^{abs''}(\mathtt{pc}_u)) \sqsubseteq \gamma^+(\Delta^{abs''}(e) \sqcup^{abs} \Delta^{abs''}(\mathtt{pc}_u))$, $\Delta''(e) \sqsubseteq \gamma^+(\Delta^{abs''}(e))$, and $\Delta''(\mathtt{pc}_u) \sqsubseteq \gamma^+(\Delta^{abs''}(\mathtt{pc}_u))$, it follows $\Delta''(e) \sqcup \Delta''(\mathtt{pc}_u) \sqsubseteq \gamma^+(\Delta^{abs''}(e) \sqcup^{abs} \Delta^{abs''}(\mathtt{pc}_u))$ or, equivalently, $\Delta'(\mathtt{pc}_u) \sqsubseteq \gamma^+(\Delta^{abs'}(\mathtt{pc}_u))$. From $\gamma^-(\Delta^{abs'}(\mathtt{pc}_u)) \sqsubseteq$

$\Delta^{abs'}(\mathtt{pc}_u) \sqsubseteq \gamma^+(\Delta^{abs'}(\mathtt{pc}_u))$, $\Delta' = \Delta''[\mathtt{pc}_u \mapsto \Delta''(e) \sqcup \Delta''(\mathtt{pc}_u)]$, $\Delta^{abs'} = \Delta^{abs''}[\mathtt{pc}_u \mapsto \Delta^{abs''}(e) \sqcup^{abs} \Delta^{abs''}(\mathtt{pc}_u)]$, and $\Delta^{abs''} \sqsubseteq_V^- \Delta'' \sqsubseteq_V^+ \Delta^{abs''}$ (see (a)), we have $\Delta^{abs'} \sqsubseteq_V^- \Delta' \sqsubseteq_V^+ \Delta^{abs'}$.
We still have to check that the requirement on C_1' and C' is satisfied. This immediately follows from $\Delta^{abs''} \sqsubseteq_V^- \Delta'' \sqsubseteq_V^+ \Delta^{abs''}$ (see (a)) and the fact that the rule adds statements of the form **set pc to** $\Delta''(\mathtt{pc}_u)$ and **set pc to** $\Delta^{abs''}(\mathtt{pc}_u)$.

- Rule F-UPDATELABELS. We can immediately apply the rule F-UPDATELABELS also according to $\rightsquigarrow$ and we get $\langle \Delta'', C_1'', M'', \langle s'', ctx''\rangle, \mathcal{S}''\rangle \xrightarrow{\mathcal{I}} \langle \Delta', C_1', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$. Then, $\Delta' = \Delta''[\mathtt{pc}_u \mapsto \ell_1]$ and $\Delta^{abs'} = \Delta^{abs''}[\mathtt{pc}_u \mapsto \ell]$. From the induction hypothesis (see (b)), we know that $\gamma^-(\ell) \sqsubseteq \ell_1 \sqsubseteq \gamma^+(\ell)$. From this, $\Delta^{abs''} \sqsubseteq_V^- \Delta'' \sqsubseteq_V^+ \Delta^{abs''}$ (see (a)), $\Delta' = \Delta''[\mathtt{pc}_u \mapsto \ell_1]$, $\Delta^{abs'} = \Delta^{abs''}[\mathtt{pc}_u \mapsto \ell]$, it follows that $\Delta^{abs'} \sqsubseteq_V^- \Delta' \sqsubseteq_V^+ \Delta^{abs'}$. Again, the requirement on C_1' and C' directly follows from the induction hypothesis (see (b)) and the fact that the rule does not introduce new statements.
- Rule F-SELECT. Most of the proof for this case is similar to that of F-ASSIGN. The only part that differs is showing that part is proving $\Delta^{abs'} \sqsubseteq_V^- \Delta' \sqsubseteq_V^+ \Delta^{abs'}$. From the rules' definitions, we have that $\Delta' = \Delta''[x \mapsto \Delta''(\mathtt{pc}_u) \sqcup L_Q(\Delta'', \varphi) \sqcup \bigsqcup_{v \in vars(\varphi)} \Delta''(v)]$ and $\Delta^{abs'} = \Delta^{abs''}[x \mapsto \Delta^{abs''}(\mathtt{pc}_u) \sqcup^{abs} L_Q^{abs}(\Delta^{abs''}, \varphi) \sqcup^{abs} \bigsqcup_{v \in vars(\varphi)} \Delta^{abs''}(v)]$. Moreover, from the induction hypothesis, we have that $\Delta^{abs''} \sqsubseteq_V^- \Delta'' \sqsubseteq_V^+ \Delta^{abs''}$. To prove that $\Delta^{abs'} \sqsubseteq_V^- \Delta' \sqsubseteq_V^+ \Delta^{abs'}$, it is enough to prove that $\gamma^-(\Delta^{abs'}(x)) \sqsubseteq \Delta'(x) \sqsubseteq \gamma^+(\Delta^{abs'}(x))$.
We first show that $\gamma^-(\Delta^{abs'}(x)) \sqsubseteq \Delta'(x)$. From $\mathcal{A}$'s soundness (requirement 3), we have $\gamma^-(\Delta^{abs''}(\mathtt{pc}_u) \sqcup^{abs} L_Q^{abs}(\Delta^{abs''}, \varphi) \sqcup^{abs} \bigsqcup_{v \in vars(\varphi)} \Delta^{abs''}(v)) \sqsubseteq \gamma^-(\Delta^{abs''}(\mathtt{pc}_u)) \sqcup \gamma^-(L_Q^{abs}(\Delta^{abs''}, \varphi)) \sqcup \bigsqcup_{v \in vars(\varphi)} \gamma^-(\Delta^{abs''}(v))$ or, equivalently, $\gamma^-(\Delta^{abs'}(x)) \sqsubseteq \gamma^-(\Delta^{abs''}(\mathtt{pc}_u)) \sqcup \gamma^-(L_Q^{abs}(\Delta^{abs''}, \varphi)) \sqcup \bigsqcup_{v \in vars(\varphi)} \gamma^-(\Delta^{abs''}(v))$. From $\Delta^{abs''} \sqsubseteq_V^- \Delta'' \sqsubseteq_V^+ \Delta^{abs''}$, we also have that $\gamma^-(\Delta^{abs''}(\mathtt{pc}_u)) \sqsubseteq \Delta''(\mathtt{pc}_u)$ and $\gamma^-(\Delta^{abs''}(v)) \sqsubseteq \Delta''(v)$ for all $v \in vars(\varphi)$. Finally, from $\mathcal{A}$'s soundness (requirement 5) and $\Delta^{abs''} \sqsubseteq_V^- \Delta'' \sqsubseteq_V^+ \Delta^{abs''}$, we have that $\gamma^-(L_Q^{abs}(\Delta^{abs''}, \varphi)) \sqsubseteq L_Q(\Delta'', \varphi)$. Hence, $\gamma^-(\Delta^{abs''}(\mathtt{pc}_u)) \sqcup \gamma^-(L_Q^{abs}(\Delta^{abs''}, \varphi)) \sqcup \bigsqcup_{v \in vars(\varphi)} \gamma^-(\Delta^{abs''}(v)) \sqsubseteq \Delta''(\mathtt{pc}_u)) \sqcup L_Q(\Delta'', \varphi) \sqcup \bigsqcup_{v \in vars(\varphi)} \Delta''(v)$. From this, $\Delta' = \Delta''[x \mapsto \Delta''(\mathtt{pc}_u) \sqcup L_Q(\Delta'', \varphi) \sqcup \bigsqcup_{v \in vars(\varphi)} \Delta''(v)]$, and $\gamma^-(\Delta^{abs'}(x)) \sqsubseteq \gamma^-(\Delta^{abs''}(\mathtt{pc}_u)) \sqcup \gamma^-(L_Q^{abs}(\Delta^{abs''}, \varphi)) \sqcup \bigsqcup_{v \in vars(\varphi)} \gamma^-(\Delta^{abs''}(v))$, we finally have that $\gamma^-(\Delta^{abs'}(x)) \sqsubseteq \Delta'(x)$.
We now show that $\Delta'(x) \sqsubseteq \gamma^+(\Delta^{abs'}(x))$. From $\mathcal{A}$'s soundness (requirement 3), we have $\gamma^+(\Delta^{abs''}(\mathtt{pc}_u)) \sqcup \gamma^+(L_Q^{abs}(\Delta^{abs''}, \varphi)) \sqcup \bigsqcup_{v \in vars(\varphi)} \gamma^+(\Delta^{abs''}(v)) \sqsubseteq \gamma^+(\Delta^{abs''}(\mathtt{pc}_u) \sqcup^{abs} L_Q^{abs}(\Delta^{abs''}, \varphi) \sqcup^{abs} \bigsqcup_{v \in vars(\varphi)} \Delta^{abs''}(v))$ or, equivalently, $\gamma^+(\Delta^{abs''}(\mathtt{pc}_u)) \sqcup \gamma^+(L_Q^{abs}(\Delta^{abs''}, \varphi)) \sqcup \bigsqcup_{v \in vars(\varphi)} \gamma^+(\Delta^{abs''}(v)) \sqsubseteq \gamma^+(\Delta^{abs'}(x))$. From $\Delta^{abs''} \sqsubseteq_V^- \Delta'' \sqsubseteq_V^+ \Delta^{abs''}$, we also have that $\Delta''(\mathtt{pc}_u) \sqsubseteq \gamma^+(\Delta^{abs''}(\mathtt{pc}_u))$ and $\Delta''(v) \sqsubseteq \gamma^+(\Delta^{abs''}(v))$ for all $v \in vars(\varphi)$. Finally, from $\mathcal{A}$'s soundness (requirement 5) and $\Delta^{abs''} \sqsubseteq_V^- \Delta'' \sqsubseteq_V^+ \Delta^{abs''}$, we have that $L_Q(\Delta'', \varphi) \sqsubseteq \gamma^+(L_Q^{abs}(\Delta^{abs''}, \varphi))$. Hence, $\Delta''(\mathtt{pc}_u) \sqcup L_Q(\Delta'', \varphi) \sqcup \bigsqcup_{v \in vars(\varphi)} \Delta''(v) \sqsubseteq \gamma^+(\Delta^{abs''}(\mathtt{pc}_u)) \sqcup \gamma^+(L_Q^{abs}(\Delta^{abs''}, \varphi)) \sqcup \bigsqcup_{v \in vars(\varphi)} \gamma^+(\Delta^{abs''}(v))$. From this, $\Delta' = \Delta''[x \mapsto \Delta''(\mathtt{pc}_u) \sqcup L_Q(\Delta'', \varphi) \sqcup \bigsqcup_{v \in vars(\varphi)} \Delta''(v)]$, and $\gamma^+(\Delta^{abs''}(\mathtt{pc}_u)) \sqcup \gamma^+(L_Q^{abs}(\Delta^{abs''}, \varphi)) \sqcup \bigsqcup_{v \in vars(\varphi)} \gamma^+(\Delta^{abs''}(v)) \sqsubseteq \gamma^+(\Delta^{abs'}(x))$, we finally have that $\Delta'(x) \sqsubseteq \gamma^+(\Delta^{abs'}(x))$.
- Rule F-UPDATEDATABASEOK. The proof of this case is similar to that of F-ASSIGN.
- Rule F-UPDATECONFIGURATIONOK. The proof of this case is similar to that of F-ASSIGN.
- The proof for the other rules is either trivial or can be derived from the above cases.

This completes the proof of our claim. $\square$

Proposition I.2 states that our security monitor equipped with a sound approximation is still secure.

Proposition I.2. *Let $\mathcal{A}$ be a sound approximation and $\rightsquigarrow_{\mathcal{A}}$ be the evaluation relation obtained by equipping our monitor with $\mathcal{A}$. For all programs $C \in Com^*_{UID}$, schedulers $\mathcal{S}$, memories $M \in Mem^*_{UID}$, and system states s, whenever $r = \langle \Delta_0^{abs}, C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\mathcal{I}}{}^n_{\mathcal{A}} \langle \Delta^{abs'}, C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$, then for all $1 \leq i \leq n$, $K_{atk}^{\rightsquigarrow_{\mathcal{A}}}(\langle M, s\rangle, C, \mathcal{S}, trace(r^{i-1})) \cap A_{atk,sec}(M, s) \subseteq K_{atk}^{\rightsquigarrow_{\mathcal{A}}}(\langle M, s\rangle, C, \mathcal{S}, trace(r^i))$, where $K_{atk}^{\rightsquigarrow_{\mathcal{A}}}$ refers to Def. 1 with $\rightsquigarrow_{\mathcal{A}}$ as evaluation relation and the system state in r's $(i-1)$-th configuration is $\langle db, U, sec, T, V\rangle$.*

Proof. We prove our claim by contradiction. Namely, assumes that the monitor defined by $\rightsquigarrow_{\mathcal{A}}$ is not sound. This happens iff there is a run $r = \langle \Delta_0^{abs}, C, M, \langle s, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\mathcal{I}}{}^n_{\mathcal{A}} \langle \Delta^{abs'}, C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$ and an index $1 \leq i \leq n$, such that $K_{atk}^{\rightsquigarrow_{\mathcal{A}}}(\langle M, s\rangle, C, \mathcal{S}, trace(r^{i-1})) \cap A_{atk,sec}(M, s) \not\subseteq K_{atk}^{\rightsquigarrow_{\mathcal{A}}}(\langle M, s\rangle, C, \mathcal{S}, trace(r^i))$, where $K_{atk}^{\rightsquigarrow_{\mathcal{A}}}$ refers to Def. 1 with $\rightsquigarrow_{\mathcal{A}}$ as evaluation relation and the system state in r's $(i-1)$-th configuration is $\langle db, U, sec, T, V\rangle$. From this, it follows that there is a global state $\langle M_1, s_1\rangle$ such that $\langle M_1, s_1\rangle \in K_{atk}^{\rightsquigarrow_{\mathcal{A}}}(\langle M, s\rangle, C, \mathcal{S}, trace(r^{i-1})) \cap A_{atk,sec}(M, s)$ but $\langle M_1, s_1\rangle \notin K_{atk}^{\rightsquigarrow_{\mathcal{A}}}(\langle M, s\rangle, C, \mathcal{S}, trace(r^i))$. This happens iff $\langle \Delta_0^{abs}, C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\mathcal{I}}{}^*_{\mathcal{A}} \langle \Delta^{abs'}, C', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$, $trace(r^{i-1}) \approx_{atk} \tau$, and $trace(r^i) \not\approx_{atk} \tau$. From Proposition I.1, it follows that $\langle \Delta_0, C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\mathcal{I}}{}^* \langle \Delta', C'', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$, $trace(r^{i-1}) \approx_{atk} \tau$, and $trace(r^i) \not\approx_{atk} \tau$. We claim that $\langle M_1, s_1\rangle \in K_{atk}^{\rightsquigarrow}(\langle M, s\rangle, C, \mathcal{S}, trace(r^{i-1}))$. From this and $\langle M_1, s_1\rangle \in A_{atk,sec}(M, s)$, we have $\langle M_1, s_1\rangle \in K_{atk}^{\rightsquigarrow}(\langle M, s\rangle, C, \mathcal{S}, trace(r^{i-1})) \cap A_{atk,sec}(M, s)$. Moreover, from $\langle \Delta_0, C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle \xrightarrow{\mathcal{I}}{}^* \langle \Delta', C'', M', \langle s', ctx'\rangle, \mathcal{S}'\rangle$ and $trace(r^i) \not\approx_{atk} \tau$, we have $\langle M_1, s_1\rangle \notin K_{atk}^{\rightsquigarrow}(\langle M, s\rangle, C,$

$\mathcal{S}, \mathit{trace}(r^i))$. Hence, $K^{\rightsquigarrow}_{atk}(\langle M, s\rangle, C, \mathcal{S}, \mathit{trace}(r^{i-1})) \cap A_{atk,sec}(M, s) \not\subseteq K^{\rightsquigarrow}_{atk}(\langle M, s\rangle, C, \mathcal{S}, \mathit{trace}(r^i))$, contradicting the soundness of our monitor (Theorem 1).

We now prove our claim that $\langle M_1, s_1\rangle \in K^{\rightsquigarrow}_{atk}(\langle M, s\rangle, C, \mathcal{S}, \mathit{trace}(r^{i-1}))$. From $\langle \Delta_0, C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle \overset{\tau}{\rightsquigarrow}^* \langle \Delta', C'', M', \langle s', \mathit{ctx}'\rangle, \mathcal{S}'\rangle$ and $\mathit{trace}(r^i) \not\approx_{atk} \tau$, it follows that there exists a trace τ'' and two distinct observations obs_1 and obs_2 such that $\mathit{trace}(r^i)\upharpoonright_{atk} = \tau''\cdot \mathit{obs}_1$ and $\tau\upharpoonright_{atk} = \tau''\cdot \mathit{obs}_2$. Moreover, $\mathit{trace}(r^{i-1})\upharpoonright_{atk} = \tau''\cdot \mathit{obs}_1$, $\mathit{trace}(r^i) \not\approx_{atk} \tau$, and $\mathit{trace}(r^{i-1}) \approx_{atk} \tau$, it follows that $\mathit{trace}(r^{i-1})\upharpoonright_{atk} = \tau''$. From this and $\tau\upharpoonright_{atk} = \tau''\cdot \mathit{obs}_2$, it follows that $\mathit{trace}(r^{i-1})\upharpoonright_{atk} \preceq \tau\upharpoonright_{atk}$. Hence, for any run $\langle \Delta_0, C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle \overset{\tau}{\rightsquigarrow}^* \langle \Delta', C'', M', \langle s', \mathit{ctx}'\rangle, \mathcal{S}'\rangle \overset{\nu}{\rightsquigarrow}^* \langle \Delta_1', C_1'', M_1', \langle s_1', \mathit{ctx}_1'\rangle, \mathcal{S}_1'\rangle$, we have that $\mathit{trace}(r^{i-1})\upharpoonright_{atk} \preceq \tau\cdot\nu\upharpoonright_{atk}$ and therefore $\mathit{trace}(r^{i-1}) \approx_{atk} \tau\cdot\nu$. Moreover, for any prefix of the run $\langle \Delta_0, C, M_1, \langle s_1, \epsilon\rangle, \mathcal{S}\rangle \overset{\tau}{\rightsquigarrow}^* \langle \Delta', C'', M', \langle s', \mathit{ctx}'\rangle, \mathcal{S}'\rangle$ we also know that the observations are consistent with $\mathit{trace}(r^{i-1})$ (this directly follows from $\mathit{trace}(r^{i-1}) \approx_{atk} \tau$). Hence, $\langle M_1, s_1\rangle \in K^{\rightsquigarrow}_{atk}(\langle M, s\rangle, C, \mathcal{S}, \mathit{trace}(r^{i-1}))$. □

I.4. Soundness of symbolic tuples

We now prove the soundness of our approximation based on symbolic tuples. Note that we split the proof in several lemmas, one per component in the approximation.

Lemma I.1. *The abstract order $\sqsubseteq^{abs}$ approximates $\sqsubseteq$.*

Proof. Let $\langle S_1^-, S_1^+\rangle, \langle S_2^-, S_2^+\rangle$ be two abstract labels such that $\langle S_1^-, S_1^+\rangle \sqsubseteq^{abs} \langle S_2^-, S_2^+\rangle$. From this, we have (1) $\forall \langle T, \varphi\rangle \in S_1^+.\ \exists \langle T, \varphi'\rangle \in S_2^-.\ \varphi \models \varphi'$, and (2) $\forall \mathit{MEM}_x \in S_1^+.\ \mathit{MEM}_x \in S_2^-$. From (1), $\forall \langle T, \varphi\rangle \in S_1^+.\ \exists \langle T, \varphi'\rangle \in S_2^-.\ \gamma(\langle T, \varphi\rangle) \subseteq \gamma(\langle T, \varphi'\rangle)$. From this and (2), it follows that $\bigcup_{\langle T,\varphi\rangle \in S_1^+} \gamma(\langle T, \varphi\rangle) \cup \{\mathit{MEM}_x \in S_1^+ \mid x \in \mathit{Var}\} \subseteq \bigcup_{\langle T,\varphi\rangle \in S_2^-} \gamma(\langle T, \varphi\rangle) \cup \{\mathit{MEM}_x \in S_2^- \mid x \in \mathit{Var}\}$. From this and property (1) of disclosure orders, $\mathit{cl}(\bigcup_{\langle T,\varphi\rangle \in S_1^+} \gamma(\langle T, \varphi\rangle) \cup \{\mathit{MEM}_x \in S_1^+ \mid x \in \mathit{Var}\}) \sqsubseteq \mathit{cl}(\bigcup_{\langle T,\varphi\rangle \in S_2^-} \gamma(\langle T, \varphi\rangle) \cup \{\mathit{MEM}_x \in S_2^- \mid x \in \mathit{Var}\})$. This is equivalent to $\gamma^+(\langle S_1^-, S_1^+\rangle) \sqsubseteq \gamma^-(\langle S_2^-, S_2^+\rangle)$. □

Lemma I.2. *The abstract join $\sqcup^{abs}$ approximates $\sqcup$.*

Proof. Let $\ell_1 = \langle S_1^-, S_1^+\rangle$ and $\ell_2 = \langle S_2^-, S_2^+\rangle$ be two abstract labels. Furthermore, let ℓ_3 be their join $\langle S_1^- \cup S_2^-, S_1^+ \cup S_2^+\rangle$. Note that, for simplicity, in the following we ignore symbols of the form MEM_x. The proof for the general case is identical.

We first show that ℓ_3 is well-defined, that is, $\gamma^-(\ell_3) \sqsubseteq \gamma^+(\ell_3)$. Assume, for contradiction's sake, that this is not the case. This follows iff there is at least one query q' in $\bigcup_{\langle T,\varphi\rangle \in S_1^- \cup S_2^-} \gamma(\langle T, \varphi\rangle)$ that is not determined by the queries in $\bigcup_{\langle T,\varphi\rangle \in S_1^+ \cup S_2^+} \gamma(\langle T, \varphi\rangle)$. Equivalently, there are two database states db and db' such that all queries in $\bigcup_{\langle T,\varphi\rangle \in S_1^+ \cup S_2^+} \gamma(\langle T, \varphi\rangle)$ produce the same results but the query q' differs. There are two cases:

- There is a symbolic tuple $\langle T, \varphi\rangle \in S_1^+$ such that $q' \in \gamma(\langle T, \varphi\rangle)$. From $\gamma^-(\langle S_1^-, S_1^+\rangle) \sqsubseteq \gamma^+(\langle S_1^-, S_1^+\rangle)$, it follows that the queries in $\bigcup_{\langle T,\varphi\rangle \in S_1^+} \gamma(\langle T, \varphi\rangle)$ determine q' (i.e., $\mathit{cl}(q') \sqsubseteq \gamma^+(\langle S_1^-, S_1^+\rangle)$). Since all queries in $\bigcup_{\langle T,\varphi\rangle \in S_1^+ \cup S_2^+} \gamma(\langle T, \varphi\rangle)$ produce the same results on db and db', then all queries in $\bigcup_{\langle T,\varphi\rangle \in S_1^+} \gamma(\langle T, \varphi\rangle)$ produce the same results on db and db'. From this and $\mathit{cl}(q') \sqsubseteq \gamma^+(\langle S_1^-, S_1^+\rangle)$, it follows that the result of q' is the same on db and db', leading to a contradiction.
- There is a symbolic tuple $\langle T, \varphi\rangle \in S_2^+$ such that $q' \in \gamma(\langle T, \varphi\rangle)$. The proof of this case is analogous.

Since both cases lead to a contradiction, it follows that ℓ_3 is well-defined, i.e., $\gamma^-(\ell_3) \sqsubseteq \gamma^+(\ell_3)$.

We now show that $\sqcup^{abs}$ approximates $\sqcup$.

First, we show that $\gamma^+(\ell_1) \sqcup \gamma^+(\ell_2) \sqsubseteq \gamma^+(\ell_3)$. By construction, $\gamma^+(\ell_1) = \mathit{cl}(\bigcup_{\langle T,\varphi\rangle \in S_1^+} \gamma(\langle T, \varphi\rangle))$, $\gamma^+(\ell_2) = \mathit{cl}(\bigcup_{\langle T,\varphi\rangle \in S_2^+} \gamma(\langle T, \varphi\rangle))$, and $\gamma^+(\ell_3) = \mathit{cl}(\bigcup_{\langle T,\varphi\rangle \in S_1^+ \cup S_2^+} \gamma(\langle T, \varphi\rangle))$. From this and $\sqcup$'s definition (property 4 of disclosure lattices), $\gamma^+(\ell_1) \sqcup \gamma^+(\ell_2)$ is equivalent to $\mathit{cl}(\bigcup_{\langle T,\varphi\rangle \in S_1^+} \gamma(\langle T, \varphi\rangle) \cup \bigcup_{\langle T,\varphi\rangle \in S_2^+} \gamma(\langle T, \varphi\rangle))$. This is, in turn, equivalent to $\mathit{cl}(\bigcup_{\langle T,\varphi\rangle \in S_1^+ \cup S_2^+} \gamma(\langle T, \varphi\rangle))$, which is exactly $\gamma^+(\ell_3)$. Hence, $\gamma^+(\ell_1) \sqcup \gamma^+(\ell_2) \sqsubseteq \gamma^+(\ell_3)$.

Second, we show that $\gamma^-(\ell_3) \sqsubseteq \gamma^-(\ell_1) \sqcup \gamma^-(\ell_2)$. By construction, $\gamma^+(\ell_1) = \mathit{cl}(\bigcup_{\langle T,\varphi\rangle \in S_1^-} \gamma(\langle T, \varphi\rangle))$, $\gamma^+(\ell_2) = \mathit{cl}(\bigcup_{\langle T,\varphi\rangle \in S_2^-} \gamma(\langle T, \varphi\rangle))$, and $\gamma^+(\ell_3) = \mathit{cl}(\bigcup_{\langle T,\varphi\rangle \in S_1^- \cup S_2^-} \gamma(\langle T, \varphi\rangle))$. As before, we can show that $\mathit{cl}(\bigcup_{\langle T,\varphi\rangle \in S_1^-} \gamma(\langle T, \varphi\rangle)) \sqcup \mathit{cl}(\bigcup_{\langle T,\varphi\rangle \in S_2^-} \gamma(\langle T, \varphi\rangle))$ is equivalent to $\mathit{cl}(\bigcup_{\langle T,\varphi\rangle \in S_1^- \cup S_2^-} \gamma(\langle T, \varphi\rangle))$. Hence, $\gamma^+(\ell_1) \sqcup \gamma^+(\ell_2)$ is equivalent to $\gamma^+(\ell_3)$, and therefore, $\gamma^+(\ell_1) \sqcup \gamma^+(\ell_2) \sqsubseteq \gamma^+(\ell_3)$. □

Lemma I.3. *The mapping $L_{\mathcal{Q}}^{abs}$ approximates $L_{\mathcal{Q}}$.*

Proof. The proposition directly follows from the results for $\ell^-_{\Delta^{abs},q}$ (Lemma I.5) and $\ell^+_{\Delta^{abs},q}$ (Lemma I.6). □

Lemma I.4. *The mapping $L_{\mathcal{Q}}^{abs}$ is exact over predicate queries.*

Proof. By construction. □

Lemma I.5. *Let $\Delta \in \Delta\!\!\Delta$, $\Delta^{abs} \in \Delta\!\!\Delta^{abs}$, and $q \in \mathcal{Q}$ be a query. If $\Delta^{abs} \sqsubseteq^-_{\mathit{supp}(q)} \Delta$, then $\gamma^-(L_{\mathcal{Q}}^{abs}(\Delta^{abs}, q)) \sqsubseteq L_{\mathcal{Q}}(\Delta, q)$.*

Proof. Let $\Delta \in \Delta\!\!\Delta$, $\Delta^{abs} \in \Delta\!\!\Delta^{abs}$, and $q \in \mathcal{Q}$ be such that $\Delta^{abs} \sqsubseteq^-_{\mathit{supp}(q)} \Delta$. We now show that $\gamma^-(L_{\mathcal{Q}}^{abs}(\Delta^{abs}, q)) \sqsubseteq L_{\mathcal{Q}}(\Delta, q)$. Note that $L_{\mathcal{Q}}(\Delta, q) = \bigsqcup_{Q \in \mathit{supp}(q)} \bigsqcup_{q' \in Q} \Delta(q')$. Given a table identifier T, we denote $\{T(\overline{v}) \in$

$\bigcup_{Q \in supp(q)} Q\}$ as $\mathbb{Q}_{q,T}$. From this, we have that $L_{\mathcal{Q}}(\Delta, q) \sqsubseteq \bigsqcup_{T \in D} \bigsqcup_{q' \in \mathbb{Q}_{q,T}} \Delta(q')$. Without loss of generality, we assume that q does not refer to views. Observe that the only interesting case is if q is a well-formed query. If that is not the case, $\ell^{-}_{\Delta^{abs},q} = \emptyset$ and, therefore, $\gamma^{-}(L^{abs}_{\mathcal{Q}}(\Delta^{abs}, q)) \sqsubseteq L_{\mathcal{Q}}(\Delta, q)$ (since $\gamma^{-}(L^{abs}_{\mathcal{Q}}(\Delta^{abs}, q)) = \bot$). Hence, assume that q is well-formed. We claim that, for each $T \in D$, $\bigcup_{\langle T,\varphi\rangle \in A_{T,q}} \gamma(\langle T, \varphi\rangle) \subseteq \mathbb{Q}_{q,T}$, where $A_q = \{\langle T, \varphi\rangle \in cstrs(q)\}$. From this, $\ell^{-}_{\Delta^{abs},q}$'s definition, and $\Delta^{abs} \sqsubseteq^{-}_{supp(q)} \Delta$, it immediately follows that $\gamma^{-}(L^{abs}_{\mathcal{Q}}(\Delta^{abs}, q)) \sqsubseteq L_{\mathcal{Q}}(\Delta, q)$.

We now prove our claim that, given a $T \in D$, $\bigcup_{\langle T,\varphi\rangle \in A_{T,q}} \gamma(\langle T, \varphi\rangle) \subseteq \mathbb{Q}_{q,T}$, where $A_q = \{\langle T, \varphi\rangle \in cstrs(q)\}$. For contradiction's sake, assume that this is not the case. Namely, there is a predicate query $T(\overline{v}) \in \bigcup_{\langle T,\varphi\rangle \in A_{T,q}} \gamma(\langle T, \varphi\rangle)$ such that $T(\overline{v}) \notin supp(q)$. For simplicity, we assume that q is of the form $\exists \overline{x}.\ T(\overline{x}) \wedge \varphi$ such that φ is satisfiable and in normal form. From this, it follows that $supp(q)$ contains all $T(\overline{v}')$ where $\overline{v}'$ satisfies φ (they are the only values that may influence q's result). Therefore, $T(\overline{v}) \in supp(q)$ and $T(\overline{v}) \in \mathbb{Q}_{q,T}$, leading to a contradiction. The reason why it is sufficient to consider queries of the form $\exists \overline{x}.\ T(\overline{x}) \wedge \varphi$ is that for more complex well-formed queries (which are boolean combinations of formulae of the form $\exists \overline{x}.\ T(\overline{x}) \wedge \varphi$) we have that $supp(\neg q) = supp(q)$, $supp(q \vee q') = supp(q) \cup supp(q')$, and $supp(q \wedge q') = supp(q) \cup supp(q')$ (this directly follows from the well-formedness requirements). □

Lemma I.6. *Let $\Delta \in \Delta\!\!\Delta$, $\Delta^{abs} \in \Delta\!\!\Delta^{abs}$, and $q \in \mathcal{Q}$ be a query. If $\Delta \sqsubseteq^{+}_{supp(q)} \Delta^{abs}$, then $L_{\mathcal{Q}}(\Delta, q) \sqsubseteq \gamma^{+}(L^{abs}_{\mathcal{Q}}(\Delta^{abs}, q))$.*

Proof. Let $\Delta \in \Delta\!\!\Delta$, $\Delta^{abs} \in \Delta\!\!\Delta^{abs}$, and $q \in \mathcal{Q}$ be such that $\Delta \sqsubseteq^{+}_{supp(q)} \Delta^{abs}$. We now show that $L_{\mathcal{Q}}(\Delta, q) \sqsubseteq \gamma^{+}(L^{abs}_{\mathcal{Q}}(\Delta^{abs}, q))$. Note that $L_{\mathcal{Q}}(\Delta, q) = \bigsqcup_{Q \in supp(q)} \bigsqcup_{q' \in Q} \Delta(q')$. Given a table identifier T, we denote $\{T(\overline{v}) \in \bigcup_{Q \in supp(q)} Q\}$ as $\mathbb{Q}_{q,T}$. From this, we have that $L_{\mathcal{Q}}(\Delta, q) \sqsubseteq \bigsqcup_{T \in D} \bigsqcup_{q' \in \mathbb{Q}_{q,T}} \Delta(q')$. We claim that, for each table identifier T such that $\langle T, \varphi\rangle \in cstrs(q)$, $\bigsqcup_{q' \in \mathbb{Q}_{q,T}} \Delta(q') \sqsubseteq \bigsqcup_{T(\overline{v}) \in \gamma(\langle T,\varphi\rangle) \cap M_T} \Delta^{abs}(T(\overline{v}))|_{+} \sqcup R(\langle T, \varphi\rangle, M_T)$. From this and $L_{\mathcal{Q}}(\Delta, q) \sqsubseteq \bigsqcup_{T \in D} \bigsqcup_{q' \in \mathbb{Q}_{q,T}} \Delta(q')$, we have $L_{\mathcal{Q}}(\Delta, q) \sqsubseteq \gamma^{+}(L^{abs}_{\mathcal{Q}}(\Delta^{abs}, q))$.

We now prove our first claim. Let T be a table identifier and $\langle T, \varphi\rangle$ be the corresponding symbolic tuple in $cstrs(q)$. Furthermore, let $T(\overline{v})$ be a tuple in $\mathbb{Q}_{q,T}$. We claim that $T(\overline{v}) \in \gamma(\langle T, \varphi\rangle)$. Therefore, there are two cases: (1) $T(\overline{v}) \in M_T$, or (2) $T(\overline{v}) \notin M_T$. In the first case, $\Delta(T(\overline{v})) \sqsubseteq \gamma(\Delta^{abs}(T(\overline{v}))|_{+})$ follows from $\Delta \sqsubseteq^{+}_{supp(q)} \Delta^{abs}$ and $T(\overline{v}) \in supp(q)$. In the second case, $\Delta(T(\overline{v})) = \Delta_0(T(\overline{v})) = cl(T(\overline{v}))$. Moreover, by construction $T(\overline{v}) \in \gamma(R(\langle T, \varphi\rangle, M_T))$. From this, $cl(T(\overline{v})) \sqsubseteq \gamma(R(\langle T, \varphi\rangle, M_T))$ and $\Delta(T(\overline{v})) \sqsubseteq \gamma(R(\langle T, \varphi\rangle, M_T))$. Hence, it follows that $\Delta(T(\overline{v})) \sqsubseteq \gamma(\Delta^{abs}(T(\overline{v}))|_{+} \sqcup^{abs} \gamma(R(\langle T, \varphi\rangle, M_T)))$ for all $T(\overline{v}) \in \mathbb{Q}_{q,T}$. Thus, $\bigsqcup_{q' \in \mathbb{Q}_{q,T}} \Delta(q') \sqsubseteq \bigsqcup_{T(\overline{v}) \in \gamma(\langle T,\varphi\rangle) \cap M_T} \Delta^{abs}(T(\overline{v}))|_{+} \sqcup R(\langle T, \varphi\rangle, M_T)$.

We now prove our second claim that whenever $T(\overline{v}) \in \mathbb{Q}_{q,T}$, then there is a symbolic tuple $\langle T, \overline{v}\rangle \in cstrs(q)$ such that $T(\overline{v}) \in \gamma(\langle T, \overline{v}\rangle)$. For simplicity, we assume that q does not refer to views (if this is not the case, we can just inline their definitions). From $T(\overline{v}) \in \mathbb{Q}_{q,T}$, it follows that there is a sub-formula $T(\overline{x}) \wedge \varphi$ occurring in q for some $\overline{x}$ and some (possibly empty) φ such that $[x_1 \mapsto \overline{v}|_1, \ldots, x_{|T|} \mapsto \overline{v}|_{|T|}]$ is a satisfying assignment for φ. From this and $cstrs(q)$'s definition, $\langle T, \varphi'\rangle \in cstrs(q)$ where $\varphi \models \varphi'$. From this and $[x_1 \mapsto \overline{v}|_1, \ldots, x_{|T|} \mapsto \overline{v}|_{|T|}]$ is a satisfying assignment for φ, it follows that $T(\overline{v}) \in \gamma(\langle T, \varphi'\rangle)$ where $\langle T, \varphi'\rangle \in cstrs(q)$. □

Lemma I.7. *The mapping $auth^{abs}$ approximates auth.*

Proof. Let sec be a policy and u be a user. Observe that $auth^{abs}(sec, u)$ is always well-defined since $\gamma^{-}(auth^{abs}(sec, u)) \sqsubseteq \gamma^{+}(auth^{abs}(sec, u))$ follows from the upper bound being $\top^{abs}$. Moreover, observe also that $cl(auth(sec, u)) \sqsubseteq \gamma^{+}(auth^{abs}(sec, u))$ also follows from the upper bound being $\top^{abs}$. Hence, we just need to show that $\gamma^{-}(auth^{abs}(sec, u)) \sqsubseteq cl(auth(sec, u))$. Let K_1 and K_2 be the set of abstract tuples $K_1 = \{\langle T, \top\rangle \mid T \in auth(s, u) \cap \mathbb{T}\}$ and $K_2 = \{\langle T, \varphi\rangle \mid V is\ a\ view \wedge V \in auth(s, u) \wedge def(V) = (T(\overline{x}) \wedge \varphi) \wedge nf(def(V))\}$. Let $T(\overline{v})$ be a concrete tuple in $\gamma(t)$ for $t \in K_1 \cup K_2$. There are two cases: (1) $t \in K_1$ and $T(\overline{v}) \in \gamma(t)$, or (2) $t \in K_2$ and $T(\overline{v}) \in \gamma(t)$. In the first case, u is authorized to read the table T in $auth(sec, atk)$. Hence, $T(\overline{v}) \in cl(\{\overline{x} \mid T(\overline{x})\}) \sqsubseteq cl(auth(sec, atk))$. In the second case, u is authorized to read a view V in $auth(sec, atk)$ of the form $T(\overline{x}) \wedge \varphi$ such that the assignment of variables to $\overline{v}$ satisfies φ. Again, $T(\overline{v}) \in cl(\{\overline{x} \mid V(\overline{x})\}) \sqsubseteq cl(auth(sec, atk))$. From this and $\gamma^{-}(auth^{abs}(sec, u)) = \bigcup_{t \in K_1 \cup K_2} \gamma(t)$, it follows that $\gamma^{-}(auth^{abs}(sec, u)) \sqsubseteq cl(auth(sec, atk))$. □

Lemma I.8. *The mapping $L^{abs}_{\mathcal{U}}$ approximates $L_{\mathcal{U}}$.*

Proof. Let s be a state, sec be the policy in s, and sec_0 be the initial policy. First, assume that $u = atk$. Then, $L^{abs}_{\mathcal{U}}(s, atk) = auth^{abs}(sec_0, atk) \sqcup^{abs} auth^{abs}(sec, atk) \sqcup^{abs} \langle\{MEM_x \mid x \in Var_{atk}\}, \top^{abs}\rangle$ and $L_{\mathcal{U}}(s, atk) = cl(auth(sec, atk)) \sqcup cl(auth(sec_0, atk)) \sqcup cl(\{MEM_x \mid x \in Var_{atk}\})$. Observe that $L^{abs}_{\mathcal{U}}(s, atk)$ is well defined since it is the result of a join of well-defined labels. Moreover, observe that $L_{\mathcal{U}}(s, atk) \sqsubseteq \gamma^{+}(L^{abs}_{\mathcal{U}}(s, atk))$ follows from the upper bound of $L^{abs}_{\mathcal{U}}$ being $\top^{abs}$. Hence, we just have to show that $\gamma^{-}(L^{abs}_{\mathcal{U}}(s, atk)) \sqsubseteq L_{\mathcal{U}}(s, atk)$ holds. From the above proposition, we know that $\gamma^{-}(auth^{abs}(sec_0, atk)) \sqsubseteq cl(auth(sec_0, atk))$ and $\gamma^{-}(auth^{abs}(sec, atk)) \sqsubseteq cl(auth(sec, atk))$. Moreover, we also have that $\gamma^{-}(\langle\{MEM_x \mid x \in Var_{atk}\}, \top^{abs}\rangle) \sqsubseteq cl(\{MEM_x \mid x \in Var_{atk}\})$. Since $\sqcup^{abs}$ approximates $\sqcup$ (see Lemma I.2), we have that $\gamma^{-}(L^{abs}_{\mathcal{U}}(s, atk)) \sqsubseteq \gamma^{-}(auth^{abs}(sec_0, atk)) \sqcup \gamma^{-}(auth^{abs}(sec, atk)) \sqcup \gamma^{-}(\langle\{MEM_x \mid x \in Var_{atk}\}, \top^{abs}\rangle)$. From this, $\gamma^{-}(auth^{abs}(sec_0, atk)) \sqsubseteq cl(auth(sec_0, atk))$, $\gamma^{-}(auth^{abs}(sec, atk)) \sqsubseteq cl(auth(sec, atk))$, and $\gamma^{-}(\langle\{MEM_x \mid x \in Var_{atk}\}, \top^{abs}\rangle) \sqsubseteq cl(\{MEM_x \mid x \in Var_{atk}\})$, it follows that $\gamma^{-}(L^{abs}_{\mathcal{U}}(s, atk)) \sqsubseteq cl(auth(sec_0, atk)) \sqcup cl(auth(sec, atk)) \sqcup cl(\{MEM_x \mid x \in Var_{atk}\})$. From this and $L_{\mathcal{U}}(s, atk) = cl(auth(sec, atk)) \sqcup cl(auth(sec_0, atk)) \sqcup cl(\{MEM_x \mid x \in Var_{atk}\})$, we have $\gamma^{-}(L^{abs}_{\mathcal{U}}(s, atk)) \sqsubseteq L_{\mathcal{U}}(s, atk)$.

Next, consider $u \neq \mathit{atk}$. Then, $L_{\mathcal{U}}^{abs}(s,u) = \langle \top^{abs}, \top^{abs} \rangle$ and $L_{\mathcal{U}}(s,u) = \top$. From $\gamma^{+}(L_{\mathcal{U}}^{abs}(s,u)) = \top$ and $\gamma^{+}(L_{\mathcal{U}}^{abs}(s,u)) = \bot$, it immediately follows that $\gamma^{-}(L_{\mathcal{U}}^{abs}(s,u)) \sqsubseteq L_{\mathcal{U}}(s,u) \sqsubseteq \gamma^{+}(L_{\mathcal{U}}^{abs}(s,u))$. □

Lemma I.9. *The abstract state* Δ_0^{abs} *is an over-approximation of* Δ_0.

Proof. It directly follows from $\Delta_0(x) = \gamma^{-}(\Delta_0^{abs}(x)) = \gamma^{+}(\Delta_0^{abs}(x))$ for any $x \in \mathit{Var} \cup \mathit{RC}^{pred} \cup \{\mathtt{pc}_u \mid u \in \mathit{UID}\}$. □

Proposition I.3. *The approximation presented in §6 is sound.*

Proof. The proposition immediately follows from Lemmas I.1–I.9. □