Information Flow Analysis for Valued-Indexed Data Security Compartments

Luísa Lourenço and Luís Caires

CITI e Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa, Portugal

Data-intensive applications as popularised by cloud computing raise many security challenges, due to the large number of remote users involved and multi-tenancy. Frequently, the security compartment associated to data stored in shared containers, such as database tables, is not determined by the static structure of the database schema, but depends on runtime data values, as required to ensure so-called "row-level" security. In this paper, we investigate a programming language approach to these issues, based on a λ-calculus extended with data manipulation primitives. We develop a type-based information flow analysis introducing a notion of value-indexed security labels, representing value-indexed security levels, or compartments. Our results ensure that well-typed programs do not break confidentiality constraints imposed by a declared security discipline.

1 Introduction

Data-centric applications have become a key component of IT infrastructures and, more recently, in the average user's daily tasks, due both to the internet's popularity and to the proliferation of cloud computing infrastructures on which most of these applications rely on nowadays. Unfortunately, such infrastructures, often based in relational database backends, do not provide enough support for the security requirements posed by such application scenarios. For instance, an app may inadvertently execute a query to extract sensitive information from the database and then insert that data in a table any user may read from, thus creating an insecure information flow, violating confidentiality. Such operation, however, may be deemed secure in a security model simply based on access control: a principal may have enough privileges to read the information and since the second table is public, he also has enough permissions to insert data there. To prevent such insecure flows, one may, at least in principle, apply to the application's code and database interface languages some form of information flow analysis [12, 14, 28].

However, data centric systems pose specific challenges on how to ensure confidentiality and integrity in the presence of multi-tenancy and container sharing. A key issue is that security compartments are not simply attached to the static structure of the database schema, but are actually dynamic and dependent on runtime data, inducing so-called "row-level" security. For a typical example, consider within a social network app a database table holding for each user the list of her private photos. We would like to consider that the security level of the whole table is (say) `photos`, but that the security level of each row is `photos(id)`, where id is the actual userid value registered on it. Assuming that different userids give rise to incomparable security levels `photos(id)` in the security lattice, we would then like to ensure that the app may never transfer private photos from one user to another, even if all photos are stored in the same database structure. A valued indexed security label such as `photos(id)` denotes a potential security compartment for each concrete `id` index value. Notice that since such indexes of security labels may be obtained by computation, or as the result of queries, they are not fixed or known at app design or construction time, and the same must be said of any indexed security label such as `photos(id)`. Nevertheless, our aim is to statically reason about information flows between the security compartments denoted by such indexed labels, so to invalidate code that break security policies. For this purpose, we introduce a type-based information flow analysis for a λ-calculus extended with a (SQL-like) data manipulation language (DML). As in classical approaches (e.g., [1, 16]), both a type τ and a security label s are assigned to expressions by our typing judgment $\Delta \vdash e : \tau^{\mathsf{s}}$, reflecting the fact that the value of e will only be affected by computations interfering at security levels $\leq \mathsf{s}$. However, we are not aware of prior work exploring value-dependent security labels, as we do here. For example, $\Delta \vdash e : \tau^{\mathsf{photos(joe)}}$ states that e will not return a photo from any user other than `joe`, even if e may read a global table containing photos of all users. Value-dependency may be explicit, as in photos(joe), or implicit, captured by security-label dependent record types, useful to express row-level security.

We discuss our approach in more detail using a toy example inspired by a healthcare center software service that manages medical doctors and patients. The system requires information about doctors and patients, as well as patient's clinical records, to be stored in a database, according to the schema

```
entity Doctors(id_d, name, age, speciality) in
 entity Patients(id_p, name, age, address) in
  entity Records(id_p, date, clinical_info) in
   entity isPatientOf(id_p, id_d) in ...
```

which can be manipulated through usual SQL-like DML primitives. We are interested in statically ensuring confidentiality properties for the applications developed over this database. Suppose that data in the healthcare system is classified in three security levels `L`, `P`, and `D` where `L` represents data that can be disclosed to the general public, `P` data that patients can see, and `D` data that only doctors can observe. As expected, a partial order for these security levels would be `L<P<D`, expressing that `L` is the most permissive level and `D` the most restrictive. To describe a data security policy, we classify the database entities fields with these security levels (we omit data types for now): **`entity`** `Doctors(id_d: L, name: L, age: D, speciality: L)`, which states that a doctor's profile (represented by a tuple in entity `Doctors`) is public except his age, that should remain secret and observable only to doctors; **`entity`** `Patients(id_p: L, name: P, age: P, address: P)`, stating a patient's profile is not public (except his identification number, which by itself does not disclose whose patient it belongs to), as well as his clinical record **`entity`** `Records(id_p: L, date: P, clinical_info: P)`, both visible to patients and to doctors (because `P<D`). A basic goal is then to guarantee that throughout the execution of any code manipulating these entities, the data stored in containers of security level `P` and `D` are not visible in a context with security level `L`, that is, this data is not made public. This may be achieved by an information flow analysis for a programming language with DML primitives, a first contribution of our work.

Yet, this basic approach is not enough to enforce the needed security policies. When we say that a patient can see his full profile we are stating something stronger than that: a patient can see *any* patient's full profile. This is clearly an undesirable limitation of the simple security label model adopted, which is not expressive enough to talk about individual tuples of an entity. In intuitive terms, the `P` security compartment needs to be partitioned (indexed) in many partitions `P(n)`, one for each patient `n`, e.g. `P(joe)`, `P(mary)`, etc, where `L`$<$`P(n)`$<$`P` for all n, and `P(m)`$\neq$`P(n)` for all $n \neq m$. Now, all patient records are stored in the same data structure, the entity table Patients. To give to the table a uniform security type, we then introduce security-label dependent records, allowing us to express“row-level" security compartments:

```
entity Doctors(id_d: L, name: L, age: D(id_d), speciality: L) in
 entity Patients(id_p:L, name:P(id_p), age:P(id_p), address:P(id_p)) in
  entity Records(id_p: L, date: P(id_p), clinical_info: P(id_p)) in
   entity isPatientOf(id_p: L, id_d: L) in ...
```

In a security-label dependent record a field identifier may act as a binding occurrence for the value it might hold, and scopes over the security labels of the remaining fields. So by indexing security level `D` with the tuple's field `id_d`, we are stating that the age info of doctor with `id_d` =`house` belongs to level `D(house)`; and likewise by indexing security level `P` with a patient's identifier `id_p`, we are saying that this data is only visible to the patient (and no other patient) and, eventually, to any doctor (since `P`$<$`D`). Value dependent security labels allows us to talk about individual tuples and capture fine grained information flows. For instance, in the following code:

```
let info = first (from (x in Records)
                  where x.id_p=42 select x.clinical_info)
 in insert [id_p:42, date:today(), clinical_info:info] in Records
```

value `info` gets security level `P(42)` since we are projecting a tuple with `id_p` value `42`. This allows the insert operation to be deemed secure since we are inserting a tuple with `id_p=42`, and `clinical_info=info` and `date=today()` with security label `P(42)` (the latter by up-classification). On the other hand, if we replace the last line above by **`insert`** `[id_p:10, date:today(), clinical_info:info]` **`in`** `Records`, an insecure flow must be signalled, since data from patient `42` is being associated to patient `10`. In other words, we prevent information from patient with `id_p` $=$ `42` to be leaked to another patient's record. Such a fine grained flow analysis is possible in our runtime value indexed security label model. Notice that even if, in this example, labels are indexed by user ids, which may stand for principals, our model is completely general, as labels may be indexed by any data (for example, one may label by `secret(doc_id)` the secret part of document `doc_id`). Moreover, our analysis is carried out for a higher-order language with a complete set of DML primitives, allowing stored procedures (higher-order store).

The main technical contributions of our work are thus the development of a type-based information flow analysis for a λ-calculus extended with DML primitives, based on a useful notion of value-dependent (indexed) security label (Section 2 and Section 3). Our technical results include type preservation and the (key) noninterference theorem, which implies that well-typed programs do not break the confidentiality constraints imposed by the declared information flow discipline (Section 4). We conclude with an overview of related work (Section 5) and a final discussion (Section 6).

$e ::=$		(*expression*)
	$\mid e.m$	(field access)
	$\mid$ **let** $x = e_1$ **in** e_2	(let)
	$\mid e_1(e_2)$	(application)
	$\mid$ **if** c **then** e_1 **else** e_2	(conditional)
	$\mid [\overline{m : e}]$	(record)
	$\mid \overline{e}$	(list)
	$\mid e_1{::}e_2$	(cons)
	$\mid$ **foreach**$(e_1, e_2, x.y.e_3)$	(iteration)
	$\mid$ **entity** $t\ (\overline{m : \tau^{\mathsf{s}}})$ **in** e	(entity)
	$\mid$ **select**$(t, x.c, x.e)$	(select)
	$\mid$ **insert**(t, e)	(insert)
	$\mid$ **update**$(t, x.e, x.c)$	(update)
	$\mid$ **delete**$(t, x.c)$	(delete)
	$\mid x$	(identifier)
	$\mid v$	(value)

$c ::=$		(*conditions*)
	$\mid \neg c$	(negation)
	$\mid c_1 \vee c_2$	(disjuction)
	$\mid V = V$	(equality)
	$\mid V$	(term)
$V ::=$		(*terms*)
	$\overline{V}$	(list)
	$\mid [\overline{m : V}]$	(record)
	$\mid \lambda(x : \tau_1^{\mathsf{s}_1}).e$	(abstraction)
	$\mid$ **true**	(true)
	$\mid$ **false**	(false)
	$\mid x$	(identifier)
	$\mid ()$	(unit)
	$\mid V.m$	(field access)
$v ::=$		(*values*)
	$\overline{v}$	(list)
	$\mid [\overline{m : v}]$	(record)
	$\mid \lambda(x : \tau_1^{\mathsf{s}_1}).e$	(abstraction)
	$\mid$ **true**	(true)
	$\mid$ **false**	(false)
	$\mid ()$	(unit)
	$\mid l$	(locations)

Fig. 1. Syntax

2 Programming Language

We carry out our development using a core programming language λ_{DB}, a typed λ-calculus with (imperative) data manipulation primitives, which we pick as a reasonable abstraction of commonly used language idioms for data centric programming [6, 8, 18]. The syntax of λ_{DB} language is given by the grammar in Fig. 1 where we assume an infinite set of *identifiers* (ranged over $x, y, t, \ldots$), an infinite set of *field names* (ranged over $s, n, m, \ldots$), and an infinite set of (store) *locations* (ranged over $l, l' \ldots$). We abbreviate indexed sets, in our syntax, with an overbar. So $[\overline{m : e}]$ stands for $[m_1{:}e_1, \ldots, m_n{:}e_n]$, $\overline{e}$ stands for $\{e_1, \ldots, e_n\}$, and `entity` $t(\overline{m{:}\tau^{\mathsf{s}}}$ `)in` e stands for `entity` $t(m_1{:}\tau_1^{\mathsf{s}_1}, \ldots, m_n{:}\tau_n^{\mathsf{s}_n}$ `)in` e. Expressions in our language are identifiers, values, field access $e.m$, application $e_1(e_2)$, abstractions $\lambda(x{:}\tau_1^{\mathsf{s}_1}).e$, constant declaration, addition of an element to a list, list iteration, conditional, and primitives for data creation and manipulation. Values include booleans, unit value $()$, collections (lists) of values $\{v_1, \ldots, v_n\}$, records $[m_1{:}v_1, \ldots,\ m_n{:}v_n]$, and abstractions $\lambda(x{:}\tau_1^{\mathsf{s}_1}).e$. These are as expected. We assume other basic data types (integers, strings) and associated operations, such as `first(-)` and `rest(-)` for collections (but omit standard details).

The list iterator primitive computes an accumulated value from the list elements. For example, to compute the sum of all the elements in a list $c = \{v_1, \ldots, v_n\}$ of integers we could write `foreach(`c`, 0, `$x.y.(x + y)$`)`. So the first expression in the list iterator primitive is the list to iterate on, and the second represents the initial value, and the third corresponds to the computation we want to perform on each iteration step. Identifiers x

and y are binding occurrences and represent an element of the list and the current value of the accumulator, respectively.

DML primitives can be explained as follows: $\texttt{\textbf{entity}}\ t\texttt{(}m_1{:}\tau_1^{\mathsf{s}_1}\texttt{,}\ldots\texttt{,}\ m_n{:}\tau_n^{\mathsf{s}_n}\ \texttt{)}\texttt{\textbf{in}}\ e$ denotes the allocation of a new database relation named t with attributes m_1 to m_n; $\texttt{\textbf{select}}\texttt{(}t\texttt{,}x.c\texttt{,}x.e\texttt{)}$ denotes the projection of a set of attributes e in a relation t for which condition c holds; $\texttt{\textbf{insert}}\texttt{(}t\texttt{,}e\texttt{)}$ denotes the insertion of a tuple denoted by expression e in the relation t; $\texttt{\textbf{update}}\texttt{(}t\texttt{,}x.e\texttt{,}x.c\texttt{)}$ denotes the replacement of each tuple in the relation t for which condition c holds by the tuple expressed by evaluating e, where x denotes the initial tuple value in c and e. Expression e is required to produce a tuple of the same type as the table, thus mentioning all of its fields. This does not limit the generality of the update primitive, since old values can be reused in the updated fields through x in e. Finally, $\texttt{\textbf{delete}}(t,x.c)$ denotes the deletion from relation t of the set of tuples for which the condition c is met. Thus in DML primitives we use the notation $x.c$ or $x.e$ to denote the condition c or body e under the entity cursor value x (which is bound in c and/or e).

For readability we sometimes use for DML primitives the natural concrete syntax adopted in Section 1. Logical conditions in syntax fragment c, used for conditionals and DML primitives, are side-effect free (pure). We distinguish a category of testable values V, used in logical conditions as terms, which denote the values v of our language plus field access and identifiers, necessary for expressing conditions in DML primitives.

The operational semantics is defined using a reduction relation. Reduction is defined between configurations of the form $(S;e)$, where S is a store, and e is an expression. A reduction step of the form $(S;e) \longrightarrow (S';e')$ states that expression e under state S evolves in one computational step to expression e' under state S'. Store S is a mapping from *locations* to collections of tuples, representing database tables. We provide an auxiliary definition for the evaluation of logical conditions.

Definition 2.1 (Conditional Expression Semantics). *The value of a closed condition c is given by the interpretation map $\mathcal{C}{:}c \rightarrow \{\textbf{true}, \textbf{false}\}$, as well as the auxiliary interpretation function for closed terms $\mathcal{T}{:}V \rightarrow v$ as follows:*

$$\mathcal{C}[\![\neg c]\!] = \neg\mathcal{C}[\![c]\!] \qquad \mathcal{C}[\![c_1 \vee c_2]\!] = \mathcal{C}[\![c_1]\!] \vee \mathcal{C}[\![c_2]\!] \qquad \mathcal{C}[\![V_1 = V_2]\!] = (\mathcal{T}[\![V_1]\!] = \mathcal{T}[\![V_2]\!])$$
$$\mathcal{T}[\![\{V_1,\ldots,V_n\}]\!] = \{\mathcal{T}[\![V_1]\!],\ldots,\mathcal{T}[\![V_n]\!]\} \qquad \mathcal{T}[\![\lambda(x:\tau_1^{\mathsf{s}_1}).e]\!] = \lambda(x:\tau_1^{\mathsf{s}_1}).e$$
$$\mathcal{T}[\![[m_1{:}V_1,\ldots,m_n{:}V_n]]\!] = [m_1{:}\mathcal{T}[\![V_1]\!],\ldots,m_n{:}\mathcal{T}[\![V_n]\!]]$$
$$\mathcal{T}[\![\textbf{true}]\!] = \textbf{true} \qquad \mathcal{T}[\![\textbf{false}]\!] = \textbf{false} \qquad \mathcal{T}[\![()]\!] = ()$$
$$\mathcal{T}[\![V.m]\!] = \mathit{field}(\mathcal{T}[\![V]\!],m) \quad \mathit{field}([\ldots,m:v,\ldots],m) = v$$

We now define our reduction relation, denoted as $(S;e) \longrightarrow (S';e')$, as being inductively defined by the rules in Fig. 2 as well as the expected rules of a call-by-value λ-calculus for the non-DML expressions. For the complete definition see [17].

Rules in Fig. 2 capture the reductions for DML primitives. Rules (*foreach-left*) and (*foreach-right*) reduce the first and second expressions of the list iterator operator, respectively. These rules, together with rule (*foreach*), imply an evaluation order from left to right. Rule (*entity*) reduces to the continuation expression e and adds a fresh location l to the store, associating it with the empty list (empty table). Rule (*select*) states that, in order to evaluate a projection operation over entity t, we need to evaluate the logical condition c and then filter out all the rows that do not satisfy the condition. The first step is achieved by first creating a list of logical conditions, $\overline{c\{{}^{r_i}/_{x}\}}$, obtained by the substitution of all free occurrences of x in conditional expression c with each record

(*foreach-left*)

$$\frac{(S; e_1) \longrightarrow (S'; e_1')}{\begin{array}{l}(S; \textbf{foreach}(e_1,\ e_2,\ x.y.e_3) \longrightarrow \\ (S'; \textbf{foreach}(e_1',\ e_2,\ x.y.e_3))\end{array}}$$

(*foreach-right*)

$$\frac{(S; e_2) \longrightarrow (S'; e_2')}{\begin{array}{l}(S; \textbf{foreach}(v,\ e_2,\ x.y.e_3) \longrightarrow \\ (S'; \textbf{foreach}(v,\ e_2',\ x.y.e_3))\end{array}}$$

(*foreach*)

$$\frac{v_l = h{::}hs}{\begin{array}{l}(S; \textbf{foreach}(v_l,\ v,\ x.y.e_3) \longrightarrow \\ (S; \textbf{foreach}(hs,\ e_3\{^{h}\!/_{x}\}\{^{v}\!/_{y}\},\ x.y.e_3))\end{array}}$$

(*foreach-base*)

$$(S; \textbf{foreach}(\{\},\ v,\ x.y.e_3) \longrightarrow (S; v)$$

(*entity*)

$$\frac{l \notin dom(S) \cup fn(e)}{\begin{array}{l}(S; \textbf{entity}\ t\ (\overline{m : \tau^{s}})\ \textbf{in}\ e) \longrightarrow \\ (S \cup \{l \mapsto \emptyset\};\ e\{^{l}\!/_{t}\})\end{array}}$$

(*select*)

$$\frac{S(l) = \{r_1, \ldots, r_n\} \quad \overline{s} = \{r_i \mid \mathcal{C}\llbracket c\{^{r_i}\!/_{x}\}\rrbracket\}}{\begin{array}{c}(S; \textbf{select}(l,\ x.c,\ x.e)) \longrightarrow \\ (S; \{e\{^{s_1}\!/_{x}\}, \ldots, e\{^{s_n}\!/_{x}\}\})\end{array}}$$

(*insert-left*)

$$\frac{(S; e) \longrightarrow (S'; e')}{(S; \textbf{insert}(l,\ e)) \longrightarrow (S'; \textbf{insert}(l, e'))}$$

(*insert*)

$$\frac{S(l) = \overline{r} \quad \overline{s} = v{::}\overline{r}}{(S; \textbf{insert}(l,\ v)) \longrightarrow (S[l \mapsto \overline{s}],\ ())}$$

(*update-init*)

$$\frac{S(l) = \{r_1, \ldots, r_n\} \quad \overline{e'} = \{r_i' \mid r_i' = (\mathcal{C}\llbracket c\{^{r_i}\!/_{x}\}\rrbracket?\ e\{^{r_i}\!/_{x}\}\ :\ r_i)\}}{(S; \textbf{update}(l,\ x.e,\ x.c)) \longrightarrow (S; \textbf{updating}(l,\ \overline{e'}))}$$

(*updating*)

$$\frac{(S; e) \longrightarrow (S'; e')}{(S; \textbf{updating}(l,\ e)) \longrightarrow (S'; \textbf{updating}(l,\ e'))}$$

(*update*)

$$\frac{}{(S; \textbf{updating}(l, v) \longrightarrow (S[l \mapsto v];\ ())}$$

(*delete*)

$$\frac{S(l) = \{r_1, \ldots, r_n\} \quad \overline{s} = \{r_i \mid \neg\mathcal{C}\llbracket c\{^{r_i}\!/_{x}\}\rrbracket\}}{(S; \textbf{delete}(l, x.c)) \longrightarrow (S[l \mapsto \overline{s}];\ ())}$$

Fig. 2. Reduction rules for DML core

r_i of entity t. Then, since for each conditional expression $c\{^{r_i}\!/_{x}\}$ corresponds a record r_i, the second step is achieved by creating a new list of expressions, $\overline{s}$, that will contain all records r_i for which their respective conditional expression $c\{^{r_i}\!/_{x}\}$ evaluated to **true**. Finally, we state that the projection operation **select** evaluates to a list of expressions obtained by replacing all free occurrences of x in e with the filtered records in $\overline{s}$. Rule (*insert-left*) reduces the expression in an insert operation while rule (*insert*) is applied when that expression is a record value, adding the value to the head of the list of records that represents entity t in the store. Rule (*update-init*) starts the computation of updated table contents by starting to compute the updating record $e\{^{r_i}\!/_{x}\}$ for each source record r_i such that the selecting condition $c\{^{r_i}\!/_{x}\}$ holds. Notice that the expression **updating**(l, e), where e is a collection expression, is not present in the source language and is only used in the operational semantics to express the (non-atomic) computation of the updated table contents, expressed by the collection e. When the the state **updating**(l, v) is reached, with a final updated collection, the table is updated in one step, using rule (*update*). Rule (*delete*) removes from the relevant entity all records r for which condition $c\{^{r}\!/_{x}\}$ holds.

$$\begin{array}{lr}
\mathsf{v} ::= & (\textit{label indexes}) \\
\quad \overline{v} & (\text{collection}) \\
\quad \mid [\overline{m : v}] & (\text{record}) \\
\quad \mid \textbf{true} & (\text{true}) \\
\quad \mid \textbf{false} & (\text{false}) \\
\quad \mid \top & (\text{top}) \\
\quad \mid \bot & (\text{bot}) \\
\quad \mid m & (\text{field identifier})
\end{array}$$

Fig. 3. Syntax of label indexes

$$\begin{array}{rlr}
\mathsf{s}, \mathsf{r}, \mathsf{t} ::= & \ell(\mathsf{v}) & \text{value dependent label} \\
\tau^{\mathsf{s}} ::= & \tau^{*\mathsf{s}} & \text{collection type} \\
& \mid (\tau^{\mathsf{s}} \to \tau'^{\mathsf{r}})^{\mathsf{t}} & \text{arrow type} \\
& \mid [\overline{m : \tau^{\mathsf{s}}}]^{\mathsf{r}} & \text{record type} \\
& \mid \mathtt{cmd}^{\mathsf{s}} & \text{command type} \\
& \mid \mathtt{Bool}^{\mathsf{s}} & \text{bool type}
\end{array}$$

Fig. 4. Syntax of types

$$(\textit{s-reflex})\ \frac{}{\tau^{\mathsf{s}} <: \tau^{\mathsf{s}}} \qquad (\textit{s-trans})\ \frac{\tau^{\mathsf{s}} <: \tau^{\mathsf{s}''} \quad \tau^{\mathsf{s}''} <: \tau^{\mathsf{s}'}}{\tau^{\mathsf{s}} <: \tau^{\mathsf{s}'}} \qquad (\textit{s-arrow})\ \frac{\tau^{\mathsf{s}'_1} <: \tau^{\mathsf{s}_1} \quad \tau'^{\mathsf{s}_2} <: \tau'^{\mathsf{s}'_2}}{(\tau^{\mathsf{s}_1} \to \tau'^{\mathsf{s}_2})^{\mathsf{t}} <: (\tau^{\mathsf{s}'_1} \to \tau'^{\mathsf{s}'_2})^{\mathsf{t}}}$$

$$(\textit{s-record})\ \frac{\forall_i\ \tau_i^{\mathsf{s}_i} <: \tau_i^{\mathsf{s}'_i} \quad \sqcup\, \mathsf{s}'_i \leq \mathsf{s}'}{[\overline{m : \tau^{\mathsf{s}}}]^{\mathsf{s}} <: [\overline{m : \tau^{\mathsf{s}'}}]^{\mathsf{s}'}} \qquad (\textit{s-expr})\ \frac{\mathsf{s} \leq \mathsf{s}'}{\tau^{\mathsf{s}} <: \tau^{\mathsf{s}'}}$$

Fig. 5. Subtyping rules

3 Type System

In this section we present our type system, which ensures the intended non-interference property. The syntax of types is defined in Fig. 4. Our types are annotated with value dependent security labels s, thus always of the form τ^{s} where τ is a standard type and s is a (possibly indexed) security label. Security labels, which may be in general value dependent, have the form $\ell(\mathsf{v})$, where v is a security label index given by the grammar in Fig. 3. A security label is *concrete* if its index is a value v, not a field identifier m. A label $\ell(m)$ where m is a field identifier only makes sense in the scope of a record type declaration, as explained below. Concrete security labels form a lattice, with $\top$ the top element (the most restrictive security level), and $\bot$ the bottom element (the most permissive security level), and $\sqcup$, $\sqcap$, denoting join and meet respectively. The lattice partial order is noted $\leq$ and $<$ its strict part; we write $\mathsf{s} \# \mathsf{s}'$ to say that neither $\mathsf{s} \leq \mathsf{s}'$ nor $\mathsf{s}' \leq \mathsf{s}$. We use $|\cdot|^{\uparrow}$ to denote the concrete upwards approximation of a label where $|\ell(m)|^{\uparrow} = \ell(\top)$ and $|\ell(v)|^{\uparrow} = \ell(v)$, and $|\cdot|^{\downarrow}$ to denote the concrete downwards approximation where $|\ell(m)|^{\downarrow} = \ell(\bot)$ and $|\ell(v)|^{\downarrow} = \ell(v)$. By convention, dependent security labels $\ell(\bot)$ and $\ell(\top)$ are interpreted as approximations to the (standard, non-value dependent) security label ℓ. We then require that for any value v, $\ell(\bot) \leq \ell(v) \leq \ell(\top)$ holds in the security lattice $\mathcal{L}$. Apart from this, we do not pose any extra assumptions of the security lattice, except that the ordering between labels is well defined and satisfies the lattice property (e.g., well defined meets and joins, etc).

Our type structure includes the boolean type, the unit type, (security-label dependent) record types, functional types, and collection types. We also assume other basic types, such as integers and strings, with associated operations, useful for examples and any practical needs. In a record type $[m_1{:}\tau_1^{\mathsf{s}_1}, \ldots, m_n{:}\tau_n^{\mathsf{s}_n}]^{\mathsf{s}'}$ any label s_i with $i > 1$ may be dependent on a previous field, and thus be of the form $\ell(m_j)$ with $j < i$. These are the only allowed occurrences of non-concrete security labels in types and programs: in the context of security-label dependent records. Moreover, for every such record type occurring in a valid typing we always have $\sqcup \mathsf{s}'_i \leq \mathsf{s}'$, where s'_i is the concrete security level of field m_i: since a record value depends on all fields, its security level cannot be lower than the level of any of its fields. The collection type $\tau^{*\mathsf{s}}$ states that each collection element has type τ^{s}, so security labels of a collection's elements are homogenous.

The typing judgment is of the form $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$, stating that expression e has type τ under typing environment Δ, and given the constraint set $\mathcal{S}$. The label s states that value of expression e does not depend on data classified with security levels above s or incomparable with s. Thus the type system ensures that only information flow from a level l to a level h such that $\mathsf{l} \leq \mathsf{h}$ is allowed. Label r expresses the security level of the computational context (cf. the "program counter" [19,22]), a standard technical device to capture implicit flows. Our type system uses a constraint system to approximate runtime values, necessary to eliminate or capture value dependencies in security labels, and approximate reasoning about runtime values. We restrict constraints to talk about *pure* expressions, without side-effects (all non-DML expressions).

Definition 3.1. *A constraint set $\mathcal{S}$ is a finite set of constraints of the form $e \doteq e'$ where e, e' are pure expressions. We assume given a decidable and sound equational theory and write $S \models e \doteq e'$ to mean that $\mathcal{S}$ entails $e \doteq e'$. We require $\doteq$ to be compatible with reduction in the sense that for any e, e' pure if $(S; e) \to (S; e')$ then $\models e \doteq e'$.*

We denote by $\mathcal{S}\{x \doteq e\}$ the set $\mathcal{S} \cup \{x \doteq e\}$ if e is a pure expression, and $\mathcal{S}$ otherwise. We give some examples of expected equational axioms:

$$\begin{array}{ll} (c \wedge c') \doteq \mathtt{true} \Rightarrow c \doteq \mathtt{true} & [\ldots, m_i{:}v_i, \ldots].m_i \doteq v_i \\ (x \doteq v) \wedge e \doteq e' \Rightarrow e\{^{v}/_{x}\} \doteq e'\{^{v}/_{x}\} & v \doteq v \end{array}$$

So, for example, $\{x.m_1 \doteq 2, x.m_1 \doteq y + 1\} \models y \doteq 1$. As for any equational theory, we assume that $S \models E$ and $\mathcal{S} \cup \{E\} \models E'$ implies $S \models E'$ (deduction closure). For our purposes, we abstract away from the particular equational theory used: the precise formulation of the equational theory is orthogonal to our approach, as long as a sound and decidable system is adopted (clearly, the more complete the theory the better). Typing declarations assign types to identifiers $x : \tau^{\mathsf{s}}$, and collection types of record types to locations, $t : \tau^{\mathsf{s}}$. A typing environment Δ is a list of typing declarations.

Definition 3.2 (Type System). *Typing is expressed by the judgment $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$, stating that expression e is well-typed by τ^{s} in environment Δ, given constraints in $\mathcal{S}$, and concrete context security level* r.

Key typing rules are given in Figs. 6 and 7, see [17] for the complete set of rules. We also define a simple subtyping relation (Fig. 5) denoted <:. Notice the role of $\leq$ in subtyping, allowing up-classification. Before discussing our typing rules, we present some useful notions related to security-label dependent record fields.

Definition 3.3 (Label approximation). *Let $\mathcal{S}$ be a constraint set, s a label, and x an identifier. We define the label approximation $\theta_x(\mathcal{S}, \mathsf{s})$ to be the concrete label given by*

$$\theta_x(\mathcal{S}, \ell(\mathsf{v})) \triangleq \textit{if}\ (\mathsf{v} = m \textit{ and } \mathcal{S} \models x.m \doteq v \textit{ for a value } v) \textit{ then } \ell(v) \textit{ else } |\ell(\mathsf{v})|^{\downarrow}.$$

$\theta_x(\mathcal{S}, \mathsf{s})$ approximates (from below) the runtime value associated to the index of the security label s, given the information provided by the constraints $\mathcal{S}$. The interesting

(*t-app*)

$$\frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : (\tau'^{\mathsf{s}'} \to \tau^{\mathsf{s}})^{\mathsf{t}} \quad \Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau'^{\mathsf{s}'}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1(e_2) : \tau^{\mathsf{s}}}$$

(*t-lambda*)

$$\frac{\Delta, x : \tau^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau'^{\mathsf{s}'}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \lambda(x : \tau^{\mathsf{s}}).e : (\tau^{\mathsf{s}} \to \tau'^{\mathsf{s}'})^{\perp}}$$

(*t-field*)

$$\frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{\mathsf{s}'} \quad \mathsf{s_i} \in \mathcal{L}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e.m_i : \tau_i^{\mathsf{s_i}}}$$

(*t-record*)

$$\frac{\forall_i \quad \Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_i : \tau_i^{\mathsf{s}_i}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} [\ldots, m_i{:}e_i, \ldots] : [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{\sqcup \mathsf{s}_i}}$$

(*t-refineRecord*)

$$\frac{\begin{array}{c}\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : [\ldots, m_j{:}\tau_j^{\mathsf{s}_j}, \ldots, m_i{:}\tau_i^{\ell_i(v_j)}, \ldots]^{\mathsf{s}} \\ \mathcal{S}\{x \doteq e\} \models x.m_j \doteq v_j\end{array}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : [\ldots, m_j : \tau_j^{\mathsf{s}_j}, \ldots, m_i : \tau_i^{\ell_i(m_j)}, \ldots]^{\mathsf{s}}}$$

(*t-id*)

$$\frac{}{\Delta, x : \tau^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}} x : \tau^{\mathsf{s}}}$$

(*t-unrefineRecord*)

$$\frac{\begin{array}{c}\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : [\ldots, m_j{:}\tau_j^{\mathsf{s}_j}, \ldots, m_i{:}\tau_i^{\ell_i(m_j)}, \ldots]^{\mathsf{s}} \\ \mathcal{S}\{x \doteq e\} \models x.m_j \doteq v_j\end{array}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : [\ldots, m_j : \tau_j^{\mathsf{s}_j}, \ldots, m_i : \tau_i^{\ell_i(v_j)}, \ldots]^{\mathsf{s}}}$$

(*t-sub*)

$$\frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}} \quad \tau^{\mathsf{s}} <: \tau^{\mathsf{s}'} \quad \mathsf{r}' \leq \mathsf{r}}{\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} e : \tau^{\mathsf{s}'}}$$

(*t-if*)

$$\frac{\begin{array}{c}\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} c : \mathtt{Bool}^{\mathsf{s}} \\ \Delta \vdash^{\mathsf{r} \sqcup \mathsf{s}}_{\mathcal{S} \cup \{c \doteq \mathtt{true}\}} e_1 : \tau^{\mathsf{s}} \\ \Delta \vdash^{\mathsf{r} \sqcup \mathsf{s}}_{\mathcal{S} \cup \{c \doteq \mathtt{false}\}} e_2 : \tau^{\mathsf{s}}\end{array}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2 : \tau^{\mathsf{s}}}$$

(*t-let*)

$$\frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : \tau^{\mathsf{s}} \quad \Delta, x : \tau^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}\{x \doteq e_1\}} e_2 : \tau'^{\mathsf{s}'}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \textbf{let } x = e_1 \textbf{ in } e_2 : \tau'^{\mathsf{s}'}}$$

Fig. 6. Typing rules for (pure) expressions (sample rules)

case is when the index v is a field identifier m_i of record x (typically, is used in typing rules for constructs where x denotes a query or update cursor of a database entity). In this case, the runtime value v may be approximated by solving $\mathcal{S} \models e \doteq v$ for v. For example, $\theta_x(\{x.m_1 \doteq 2, x.m_2 \doteq x.m_1 + 3\}, \ell(m_2)) = \ell(5)$ because $\{x.m_1 \doteq 2, x.m_2 \doteq x.m_1 + 3\} \models x.m_2 \doteq 5$. But $\theta_x(\{x.m_1 \doteq 2, x.m_2 \doteq x.m_1 + 3\}, \ell(m_3)) = \ell(\perp)$ since there is no v such that $\{x.m_1 \doteq 2, x.m_2 \doteq x.m_1 + 3\} \models x.m_3 \doteq v$.

We now discuss our key typing rules, omitting the standard ones for any typed λ-calculus. Rule (*t-refineRecord*) can be used to introduce a field dependent security label for a field m_j, given that we can derive that the indexing field contains a runtime value v_j (via relation $\models$). The converse is achieved with rule (*t-unrefineRecord*). In Fig. 7 we present the typing rules for DML primitives. Type rule (*t-entity*) adds the entity's identifier to the type context, assigning to it a collection type of record types where each field's type corresponds to the ones declared, and types the continuation expression with the extended typing context. To approximate the table record security level, we use $|\mathsf{s}|^{\uparrow}$.

In rule (*t-select*), to prevent implicit flows, we force the security label of the conditional expression to be the same as the one for body expression e and change the computational context label r to the least upper bound between the current context security label and the conditional expression's label. Rule (*t-insert*) states that an insert primitive is well-typed if the expression has exactly the same security type as the entity's elements, this prevents explicit flows from occurring. Moreover, in order to prevent im-

(*t-foreach*)

$$\frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : \tau^{*\mathsf{s}} \quad \Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau'^{\mathsf{s}} \quad \Delta, x : \tau^{\mathsf{s}}, y : \tau'^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}} e_3 : \tau'^{\mathsf{s}}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \textbf{foreach}\ (e_1,\ e_2,\ x.y.e_3) : \tau'^{\mathsf{s}}}$$

(*t-entity*)

$$\frac{\Delta, t : [m_1 : \tau_1^{\mathsf{s}_1}, \ldots, m_n : \tau_n^{\mathsf{s}_n}]^{*\sqcup|\mathsf{s}_i|^{\uparrow}} \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}'}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \textbf{entity}\ t(m_1 : \tau_1^{\mathsf{s}_1}, \ldots, m_n : \tau_n^{\mathsf{s}_n})\ \textbf{in}\ e : \tau^{\mathsf{s}'}}$$

(*t-select*)

$$\frac{\begin{array}{c}\Delta(t) = [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{*\mathsf{s}} \quad \mathcal{S}' = \mathcal{S} \cup \{c \doteq \mathtt{true}\} \\ \Delta, x : [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}} c : \mathtt{Bool}^{\mathsf{s}'} \\ \Delta, x : [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{\mathsf{s}} \vdash^{\mathsf{r}\sqcup\mathsf{s}'}_{\mathcal{S}'} e : \tau^{\mathsf{s}'}\end{array}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \textbf{select}(t,\ x.c,\ x.e) : \tau^{*\mathsf{s}'}}$$

(*t-insert*)

$$\frac{\begin{array}{c}\Delta(t) = [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{*\mathsf{s}} \\ \Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{\mathsf{s}} \quad \forall_i\ \mathsf{r} \leq \theta_x(\mathcal{S}\{x \doteq e\}, \mathsf{s}_i)\end{array}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \textbf{insert}(t,\ e) : \mathtt{cmd}^{\perp}}$$

(*t-update*)

$$\frac{\begin{array}{c}\Delta(t) = [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{*\mathsf{s}} \quad \mathcal{S}' = \mathcal{S} \cup \{c \doteq \mathtt{true}\} \\ \Delta, x : [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}} c : \mathtt{Bool}^{\mathsf{s}'} \\ \Delta, x : [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}'} e : [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{\mathsf{s}'} \\ \forall_i\ \mathsf{r} \sqcup \mathsf{s}' \leq \theta_x(\mathcal{S}', \mathsf{s}_i) \sqcap \theta_y(\mathcal{S}\{y \doteq e\}, \mathsf{s}_i)\end{array}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \textbf{update}(t,\ x.e,\ x.c) : \mathtt{cmd}^{\perp}}$$

(*t-delete*)

$$\frac{\begin{array}{c}\Delta(t) = [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{*\mathsf{s}} \\ \Delta, x : [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}} c : \mathtt{Bool}^{\mathsf{s}'} \\ \forall_i\ \mathsf{r} \sqcup \mathsf{s}' \leq \theta_x(\mathcal{S} \cup \{c \doteq \mathtt{true}\}, \mathsf{s}_i)\end{array}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \textbf{delete}(t,\ x.c) : \mathtt{cmd}^{\perp}}$$

Fig. 7. Typing rules for DML core

plicit flows, we check if the current computational context is lower than or equal to all the fields' security labels. But since these labels may mention field identifiers, we use θ_x to approximate the concrete levels (x fresh in the conclusion). In rules (*t-update*) and (*t-delete*) we apply the same principles of rule (*t-insert*) to prevent explicit flows, and of rules (*t-select*) and (*t-insert*) to account for implicit flows. In rule (*t-update*) one needs to make sure that the least upper bound between the current context security label ("program counter" security level) and the conditional expression's label is below or equal to the greatest lower bound of the security levels of both the "old" records and "updated" records, so to make sure only up-flows are allowed. In (*t-update*), y is required fresh in the conclusion. As already mentioned, expression e is required to produce a tuple of the same type as the table elements, updating all fields. This does not limit the generality of the update primitive, since the old values may be used in the updated fields through x in e and is in fact convenient for the expressiveness of our information flow analysis.

4 Type Preservation and Non-Interference

In this section we present the soundness results for our type system, which ensures that well-typed programs do not leak confidential information to security compartments unrelated by the assumed security lattice. Main statements are Theorem 4.1 (Type Preservation) - types are preserved by the reduction relation - Theorem 4.2 (Progress) - well-typed expressions are either a value or have a reduction step - and 4.3 (Noninterference) - well-typed expressions satisfy the noninterference property.

We now introduce notions of store consistency and well-typed configurations, and define an expression equivalence relation.

Definition 4.1 (Store Consistency). *Given a typing environment Δ and a store S, we say store S is consistent with respect to typing environment Δ, denoted as $\Delta \vdash S$, if $dom(S) \subseteq dom(\Delta)$ and $\forall l \in dom(S)$ then $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} S(l) : \Delta(l)$.*

$$\frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1} v : \tau^{\mathsf{s}'} \quad \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_2} v : \tau^{\mathsf{s}'}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} v : \tau^{\mathsf{s}'}}\ (\textit{e-val}) \qquad \frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1} v_1 : \tau^{\mathsf{s}'} \quad \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_2} v_2 : \tau^{\mathsf{s}'} \quad \mathsf{s} < \mathsf{s}'}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2 : \tau^{\mathsf{s}'}}\ (\textit{e-valOpaque})$$

$$\frac{\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \tau^{\mathsf{s}''} \quad \tau^{\mathsf{s}''} <: \tau^{\mathsf{s}'} \quad \mathsf{r} \leq \mathsf{r}'}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \tau^{\mathsf{s}'}}\ (\textit{e-sub})$$

$$\frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : [\ldots, m_j : \tau_j^{\mathsf{s}_j}, \ldots, m_i : \tau_i^{\ell_i(v_j)}, \ldots]^{\mathsf{s}'} \quad \mathcal{S}_1\{x \doteq e\} \models x.m_j \doteq v_j \quad \mathcal{S}_2\{x \doteq e'\} \models x.m_j \doteq v_j}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : [\ldots, m_j : \tau_j^{\mathsf{s}_j}, \ldots, m_i : \tau_i^{\ell_i(m_j)}, \ldots]^{\mathsf{s}'}}\ (\textit{e-refineRecord})$$

Fig. 8. Equivalence of expressions up to level s (sample rules)

Definition 4.2 (Well-typed Configuration). *A configuration $(S; e)$ is well-typed in Δ if $\Delta \vdash S$ and $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$.*

To prove type preservation, we introduce the substitution lemma on which it relies.

Lemma 4.1. *If $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v{:}\tau'^{\mathsf{s}'}$ and $\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e{:}\tau^{\mathsf{s}}$ then $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e\{v/x\}{:}\tau^{\mathsf{s}}$.*
Proof: Induction on the derivation of $\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e{:}\tau^{\mathsf{s}}$, see [17].

Theorem 4.1 states that any reduction step of a well-typed configuration leads to a well-typed configuration, where the typing of the final configuration is possibly extended with new locations of entities (via **`entity`**) in the state.

Theorem 4.1. *Let $\Delta \vdash S$ and $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e{:}\tau^{\mathsf{s}}$. If $(S; e) \longrightarrow (S'; e')$ then there is Δ' such that $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e' : \tau^{\mathsf{s}}$, $\Delta' \vdash S'$ and $\Delta \subseteq \Delta'$.*
Proof: Induction on the derivation of $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e{:}\tau^{\mathsf{s}}$, see [17].

The next result says that in a well-typed configuration $(S; e)$ either e is a value, or e has a reduction step. Theorem 4.2 says that well-typed programs never get stuck.

Theorem 4.2. *Let $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e{:}\tau^{\mathsf{s}}$ and $\Delta \vdash S$. If e is not a value then $(S; e) \longrightarrow (S'; e')$.*
Proof: Induction on the derivation of $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$, see [17].

In order to formulate our main result (non-interference), we need to present some auxiliary definitions, namely store equivalence and expression equivalence up to a security level. To define store equivalence we use the *filter* function *filter*(Δ, S, s) that given store S returns the store obtained from S by redacting (replace by $\star$) all stored values classified at levels above security level s or incomparable to s (see [17]). Stores S_1, S_2 such that $\Delta \vdash S_i$ are said equivalent up to level s if *filter*$(\Delta, S_1, \mathsf{s}) =$ *filter*$(\Delta, S_2, \mathsf{s})$.

Definition 4.3 (Expression Equivalence). *Expression equivalence of e_1 and e_2 up to s is asserted by $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2{:}\tau^{\mathsf{s}'}$.*

Intuitively, two expressions are equivalent up to level s if they yield the same result under stores equivalent up to s. Such formulations of equivalence are usual for expressing non-interference, and always relate expressions at the same type and security level. Notice that two expressions may be equivalent up to level s even if they are classified at a different level s'. Technically, expressions e_1 and e_2 are equivalent up to level s if they only differ in subexpressions classified at higher (or incomparable) security levels (so they cannot be distinguished by attackers constrained to see only up to level s). We show key defining rules for expression equivalence in Fig. 8 (full set of rules in [17]).

We can, at last, present our final main result: the non-interference theorem.

Theorem 4.3. *Let $\Delta \vdash S_1$, $\Delta \vdash S_2$, $S_1 =_{\mathsf{s}} S_2$, and $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2 : \tau^{\mathsf{s}'}$. If $(S_1, e_1) \longrightarrow (S_1', e_1')$, and $(S_2, e_2) \longrightarrow (S_2', e_2')$ then there is Δ' such that $\Delta \subseteq \Delta'$, $\Delta' \vdash S_1'$, $\Delta' \vdash S_2'$, $S_1' =_{\mathsf{s}} S_2'$, and $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1' \cong_{\mathsf{s}} e_2' : \tau^{\mathsf{s}'}$.*

Proof: Induction on the derivation of $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2 : \tau^{\mathsf{s}'}$, see [17].

The non-interference states that if we execute two equivalent instances of the same program under stores that differ only on information with higher (or incomparable) security level than s, then the resulting stores remain indistinguishable up to security level s. As a consequence we have

Corollary 4.1. *Assume $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e{:}\tau^{\mathsf{s}'}$, $\Delta \vdash S_1$, $\Delta \vdash S_2$ and $S_1 =_{\mathsf{s}} S_2$. If $(S_1, e) \xrightarrow{*} (S_1', v_1)$, and $(S_2, e) \xrightarrow{*} (S_2', v_2)$ then there is Δ' such that $\Delta \subseteq \Delta'$, $\Delta' \vdash S_1'$, $\Delta' \vdash S_2'$, $S_1' =_{\mathsf{s}} S_2'$ and $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2{:}\tau^{\mathsf{s}'}$. Also, for all s if $\mathsf{s} \not\leq \mathsf{s}'$ then $v_1 = v_2$.*

The above result precisely express the fact that an attacker "located" at security level s cannot distinguish between evaluations of a program taking place in environments that only differ in data that should be considered confidential for level s, e.g., data classified at any level l such that $\mathsf{l} \not\leq s$. To illustrate the non-interference theorem, we use **entity** `Records (id_p:int`$^{\mathsf{L}}$`, date:Date`$^{\mathsf{P(id\_p)}}$`, clinical_info:tinfo`$^{\mathsf{P(id\_p)}}$`)`.

```
let info= first( from (x in Records)
                 where x.id_p=42 select x.clinical_info)
 in insert [id_p:42, date:today(), clinical_info:info] in Records
```

Assume P(10)#P(42). Let S_1 and S_2 be stores such that S_1(Records) $= \{$(10,d$_1$,X), (42,d$_2$, A)$\}$ and S_2(Records) $= \{$(10,d$_1$,X), (42,d$_2$,B)$\}$ with $A \neq B$. We have $S_1 =_{P(10)} S_2$ since the values A and B, classified as P(42), are not visible at level P(10). We have $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2}$ **insert** [id_p: 42, date: today(), clinical_info: info] **in** Records $\cong_{P(10)}$ **insert** [id_p: 42, date: today(), clinical_info: info] **in** Records: `cmd`$^{\perp}$. Consider the reductions (S_1;**insert** [id_p: 42, date: today(), clinical_info: info] **in** Records)$\longrightarrow$(S_1'; ()) and (S_2;**insert** [id_p: 42,date:today(), clinical_info:info] **in** Records) $\longrightarrow$ (S_2'; ()). Then S_1'(Records) $= \{$(10,d$_1$,X), (42,d$_2$,A), (42, d$_3$,A)$\}$ and S_2'(Records) $= \{$(10,d$_1$,X), (42,d$_2$,B), (42,d$_3$,B)$\}$ meaning the property is satisfied, since *filter*(Δ, S_1',P(10)) $= \{$(10,d$_1$,X), (42,$\star$,$\star$), (42, $\star$,$\star$)$\}$ $=$ *filter*(Δ, S_2',P(10)), that is $S_1' =_{P(10)} S_2'$. So, the effects of the above command are not visible at security level P(10). Now consider the following slight modification of the code above:

```
let info= first (from (x in Records)
                 where x.id_p=42 select x.clinical_info)
 in insert [id_p: 10, date: today(), clinical_info: info] in Records
```

We have $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2}$ **insert** [id_p: 10, date: today(), clinical_info: info] **in** Records$\cong_{P(10)}$ **insert** [id_p: 10, date: today(), clinical_info: info] **in** Records: `cmd`$^{\perp}$. After reduction steps (S_1;**insert** [id_p:10, date: today(), clinical_info:info] **in** Records)$\longrightarrow$(S_1'; ()) and (S_2;**insert** [id_p:10, date: today(), clinical_info:info] **in** Records)$\longrightarrow$ (S_2'; ()) we have S_1'(Records)$=\{$(10,d$_1$,X), (42,d$_2$, A), (10,d$_3$,A)$\}$ and S_2'(Records)$=\{$(10,d$_1$,X), (42,d$_2$,B), (10,d$_3$,B)$\}$. But now $S_1' \neq_{P(10)} S_2'$

since, upon the insertion of the tuple [id_p: 10, date: today(), clinical_info: info] , the values A and B are observable at level P(10). This is captured by the notion of store equivalence since $\mathit{filter}(\Delta, S'_1,\text{P(10)}) = \{(10,\text{d}_1,\text{X}), (42,\star,\star), (10, \text{d}_3,\text{A})\}$ and $\mathit{filter}(\Delta, S'_2, \text{P(10)}) = \{(10,\text{d}_1,\text{X}), (42,\star,\star), (10, \text{d}_3,\text{B})\}$. Thus, the thesis of the noninterference theorem is not satisfied, and this program is rejected by the type system. Clearly, it is not possible to give the appropriate (security-label dependent record) type [id_p:int$^{\text{L}}$, date:Date$^{\text{P(id\_p)}}$, clinical_info:tinfo$^{\text{P(id\_p)}}$] to the record [id_p:10, date: today(), clinical_info:info] (using rule (*t-unrefineRecord*)) since the security level of info is P(42) but the value of id_p is 10.

5 Related Work

Our core typed λ-calculus equipped with SQL-like DML primitives is inspired in proposals such as [5, 9, 18] and provides a natural vehicle to investigate information flow security analyses for data-centric software systems. There is a huge body of work on language-based information flow analysis (see e.g., [22]). Early works [23,28] focus on simple imperative languages. [16] presents an information flow analysis for a λ-calculus with references, sums, and product types. The dependency calculus [1] generalises [16] and is shown to be able to encode the languages of [16, 28], as well as the respective noninterference results.

Several proposals for information flow analysis on web-based or data-centric applications were put forward. In [2] a dynamic information flow mechanism for a Javascript-like language based on a notion of faceted values is proposed. Faceted values offer different views of a value given the execution context's principal. Other recent work [15] proposes a dynamic information flow analysis for a subset of the ECMA standard for Javascript. In [13] a taint analysis for mobile applications is proposed, where implicit flows are not taken into account to minimise performance overhead, and in [11] a dynamic analysis to prevent insecure cross-application information flows is developed. Other works based on dynamic analysis include [7, 24, 30]. Our work is instead based on static analysis, as we seek to obtain compile time security guarantees, and avoid possible information leaks due to exceptional behavior (dynamic security errors).

Several works on static language-based analysis of information flow policies for web and data-centric software systems have also been proposed. The (seminal) decentralised label model of [20] supports static analysis of information flow policies based on labelling data elements with policies that specifies allowed reader and writer principals, but which cannot easily express value-dependent "row-level" security compartments as we do. In [10] the authors present a static analysis to enforce label-based security policies in the web programming language SELinks. Their analysis is able to enforce relevant information flow policies in web applications although the authors do not discuss the noninterference property. Examples of other works aiming to statically enforce data security on DML-based applications are [8] and [5]. The approach taken by the former consists in adding program specifications expressed by SQL-queries which are then typechecked, while the latter uses refinement types and semantic subtyping to enforce properties that may be relevant for security. Unlike our approach, these works do not provide a value-dependent information flow analysis leading to non-interference results, as we have done here. An interesting work tackling dynamic issues within an statically verified information flow framework is [27], which addresses the manipulation of runtime first-class representations of principals. In [31] an information flow analysis is introduced where security labels can actually be changed at runtime and case-analysed by a conditional-construct. In our work we do not consider dynamically changing labels but, instead, use runtime values to index security labels to ensure “row-level" security. In [6] a refinement type-based approach for data security for a DML, which can handle policies depending on the database state, is proposed. To some extent, the work in this paper can be seen as recasting some ideas of [6] in the setting of a much more expressive information flow analysis framework. Several recent works explore applications of refinement types [3, 4] and dependent types [25, 26] to language-based security, however we are not aware of prior work exploring value-dependent security labels within an information flow framework, in the sense introduced in this paper.

6 Concluding Remarks

We have developed an expressive type-based information flow analysis for a core λ-calculus augmented with SQL-like primitives. Our core language may be seen as a convenient abstraction for practical data-manipulating programming languages. A key novelty of our work consists in the introduction of a notion of value-indexed security label, which allows us to parametrize security compartments on the values of computations or on dynamically stored data, thus enabling the enforcement of so-called “row-level" security policies within shared containers, relevant to address several realistic security concerns of data-centric software services. Our main technical results are type preservation and noninterference theorems, which ensure the soundness of our information flow analysis: well-typed programs do not disclose information for security compartments unauthorised by the security lattice.

As information flow analysis per se is not enough in general to provide complete security guarantees [21], we would like to combine our value dependent type-based information flow analysis with a suitable form of role-based access control. Some of the techniques used in this paper relate to the general notion of dependent types. It certainly deserves further study and generalisation, namely regarding its connection with more standard dependent type systems [26, 29]. In particular, the introduction of first-order dependent function types in our language, which would add to the expressiveness of our approach, should be further investigated.

References

1. M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke:. A Core Calculus of Dependency. In *POPL'99*, 1999.
2. T. H. Austin and C. Flanagan. Multiple Facets for Dynamic Information Flow. In *POPL'12*, 2012.
3. I. G. Baltopoulos, J. Borgström, and A. D. Gordon. Maintaining Database Integrity with Refinement Types. In *ECOOP 2011*. Springer, 2011.
4. J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement Types for Secure Implementations. *ACM Trans. Program. Lang. Syst.*, 2011.
5. G. M. Bierman, A. D. Gordon, C. Hritcu, and D. E. Langworthy. Semantic Subtyping with an SMT Solver. *J. Funct. Program.*, 2012.
6. L. Caires, J. A. Pérez, J. C. Seco, H. T. Vieira, and L. Ferrão. Type-based Access Control in Data-Centric Systems. In *ESOP 2011*, LNCS. Springer-Verlag, 2011.

7. W. Cheng, D. Ports, D. Schultz, V. Popic, A. Blankstein, J. Cowling, D. Curtis, L. Shrira, and B. Liskov. Abstractions for Usable Information Flow Control in Aeolus. In *USENIX Annual Technical Conference*, 2012.
8. A. Chlipala. Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications. In *USENIX OSDI 2010*. USENIX Association, 2010.
9. E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web Programming Without Tiers. In *FMCO 2006*. Springer, 2007.
10. B. J. Corcoran, N. Swamy, and M. W. Hicks. Cross-tier, Label-based Security Enforcement for Web Applications. In *ACM SIGMOD Int. Conf. on Management of Data*. ACM, 2009.
11. B. Davis and H. Chen. DBTaint: Cross-Application Information Flow Tracking via Databases. In *USENIX WebApps'10*, 2010.
12. D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Comm. of the ACM*, 1977.
13. W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. M., and A. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *USENIX OSDI 2010*. USENIX Association, 2010.
14. J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *IEEE SSP'82*, 1982.
15. D. Hedin and A. Sabelfeld. Information-Flow Security for a Core of JavaScript. In *CSF 2012*. IEEE, 2012.
16. N. Heintze and J. G. Riecke. The SLam Calculus: Programming with Secrecy and Integrity. In *POPL'98*, 1998.
17. L. Lourenço and L. Caires. Information Flow Analysis for Valued-Indexed Data Security Compartments. Technical report, UNL, 2013. `http://ctp.di.fct.unl.pt/~luisal/resources/techreportDLIF13.pdf`.
18. E. Meijer, B. Beckman, and G. M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD Int. Conf. on Management of Data*. ACM, 2006.
19. A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *POPL'99*, 1999.
20. A. C. Myers and B. Liskov. A Decentralized Model for Information Flow Control. In *SOSP*, 1997.
21. A. Nanevski, A. Banerjee, and D. Garg. Verification of Information Flow and Access Control Policies with Dependent Types. In *IEEE SSP'11*, 2011.
22. A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications, special issue on Formal Methods for Security*, 2003.
23. A. Sabelfeld and D. Sands. A Per Model of Secure Information Flow in Sequential Programs. *Higher-Order and Symbolic Computation*, 2001.
24. D. Schultz. *Decentralized Information Flow Control for Databases*. Ph.D., MIT, 2012.
25. N. Swamy, J. Chen, and R. Chugh. Enforcing Stateful Authorization and Information Flow Policies in Fine. In *ESOP 2010*. Springer, 2010.
26. N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang. Secure Distributed Programming with Value-dependent Types. In *ICFP 2011*. ACM, 2011.
27. S. Tse and S. Zdancewic. Run-time Principals in Information-flow Type Systems. *ACM Trans. Program. Lang. Syst.*, 2007.
28. D. M. Volpano, C. E. Irvine, and G. Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 1996.
29. H. Xi and F. Pfenning. Dependent Types in Practical Programming. In *POPL '99*. ACM, 1999.
30. N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing Distributed Systems with Information Flow Control. In *USENIX NSDI 2008*. USENIX Association, 2008.
31. L. Zheng and A. C. Myers. Dynamic Security Labels and Static Information Flow Control. *Int. J. Inf. Sec.*, 2007.