Information-flow Analysis of Hibernate Query Language

Agostino Cortesi[1] and Raju Halder[2]

[1] Università Ca' Foscari Venezia, Italy, cortesi@unive.it
[2] Indian Institute of Technology Patna, India, halder@iitp.ac.in

Abstract. Hibernate Query Language (HQL) provides a framework for mapping object-oriented domain models to traditional relational databases. In this context, existing information leakage analyses cannot be applied directly, due to the presence and interaction of high-level application variables and SQL database attributes. The paper extends the Abstract Interpretation framework to properly deal with this challenging applicative scenario, by using the symbolic domain of positive propositional formulae to capture variable dependences affecting (directly or indirectly) the propagation of confidential data.

Key words: Hibernate Query Language, Information Leakage, Static Analysis, Abstract Interpretation

1 Introduction

Hibernate Query Language (HQL) provides a framework for mapping object-oriented domain models to traditional relational databases [1, 2, 6]. Basically it is an ORM (Object Relational Mapping) which solves object-relational impedance mismatch problems, by replacing direct persistence-related database accesses with high-level object handling functions. Various methods in "`Session`" are used to propagate object's states from memory to the database (or vice versa). Hibernate will detect any change made to an object in persistent state and synchronizes the state with the database when the unit of work completes. A HQL query is translated by Hibernate into a set of conventional SQL queries during run time which in turn performs actions on the database.

Preserving confidentiality of sensitive information in software systems always remains a thrust area for researchers. Sensitive data may be leaked maliciously or even accidentally through a bug in the program [14]. For example, any health information processing system may release patient's data, or any online transaction system may release customer's credit card information through covert channels while processing.

The following code fragments depict two different scenarios (explicit/direct flow and implicit/indirect flow) of information leakage:

`Explicit Flow`	`Implicit flow`

`l := h`	`if(h==0)`
	`    l=5;`
	`else`
	`    l=10;`

Assuming variables 'h' and 'l' are private and public respectively, it is clear from the code that confidential data in 'h' can be deduced by attackers observing 'l' on the output channel.

As traditional security measures (*e.g.* access control, encryption, etc.) do not fit to solve this when sensitive information is released from the source legitimately and it is propagated through the software during computations, various language-based information flow security analysis approaches are proposed [9, 10, 14, 15]. This is formalized by the non-interference principle that says "a variation of confidential data does not cause any variation to public data". Works in this direction have been starting with the pioneering work of Dennings in the 1970s [5].

Most of the notable works [8–10, 13] which refer to imperative, object-oriented, functional, and structured query languages, can not be applied directly to the case of HQL due to the presence and interaction of high-level HQL variables and database attributes through `Session` methods. Moreover, analyzing object-oriented features of HQL does not meet our objectives neither.

In this paper, we extend the abstract interpretation-based framework in [16] to the case of HQL, focussing on `Session` methods which act as persistent manager. This allows us to perform leakage analysis of sensitive database information when is accessed through high-level HQL code.

The proposed approach is two-folded:

- Defining the concrete and an abstract transition semantics of HQL, by using symbolic domain of positive propositional formulae.
- Analyzing possible information leakage based on the abstract semantics, focussing on variable dependences of database attributes on high-level HQL variables.

The structure of the paper is as follows: Section 2 provides a motivational example. In Section 3, we formalize the concrete and an abstract transition semantics of HQL, by using the symbolic domain of positive propositional formulae. In Section 4, we perform information leakage analysis of programs based on the abstract semantics which captures possible leakage of confidential data. Section 5 concludes the paper.

2 Motivating Example

The language-based information flow security analysis has been applied in case of object-oriented languages, aiming at identifying possible information leakage to unauthorized users [9, 10, 12]. However, the conventional approaches do not fit to the case of HQL, when considering the sensitivity level of database information and influence on them through high-level HQL variables.

Consider, for instance, an example in Figure 1(a). Here, values of the table corresponding to the class c_1 are used to make a list, and for each element of the list an update is performed on the table corresponding to the class c_2. Observe that there is an information-flow from confidential (denoted by h) to public variables (denoted by l). In fact, the confidential database information h_1 which is extracted at statement 3, affects the public view of the database information l_1 at statement 8. This fact is depicted in Figure 1(b).

The new challenge in this scenario *w.r.t.* state-of-the-art of information leakage detection is that we need to consider both application variables and SQL variables (corresponding to the database attributes).

3 Concrete and Abstract Semantics of HQL

We refer to the semantics of object-oriented programming language as defined in [11]. We just recall some basics of it. Then we formalize the concrete and abstract transition semantics of HQL, considering the Hibernate `Session` Objects, in order to identify possible information leakage.

3.1 Concrete Semantics

Object-Oriented Programming (OOP) language consists of a set of classes including a main class from where execution starts. Therefore, a program P in OOP is defined as $P = \langle c_{main}, \texttt{L}\rangle$ where `Class` denotes the set of classes, $c_{main} \in \texttt{Class}$ is the main class, $\texttt{L} \subset \texttt{Class}$ are the other classes present in P. A class $c \in \texttt{Class}$ is defined as a triplet $c = \langle \texttt{init}, \texttt{F}, \texttt{M}\rangle$ where `init` is the constructor, `F` is the set of fields, and `M` is the set of member methods in c.

Let `Var`, `Val` and `Loc` be the set of variables, the domain of values and the set of memory locations respectively. The set of environments is defined as $\texttt{Env} : \texttt{Var} \longrightarrow \texttt{Loc}$. The set of stores is defined as $\texttt{Store} : \texttt{Loc} \longrightarrow \texttt{Val}$.

The semantics of constructor and methods are defined below. Given a store s, the constructor maps its fields to fresh locations and then assigns values into those locations. Constructors never return output, but methods may return output.

Definition 1 (Constructor Semantics). *Given a store s. Let $\{a_{in}, a_{pc}\} \subseteq \texttt{Loc}$ be the free locations, $\texttt{Val}_{in} \subseteq \texttt{Val}$ be the semantic domain for input values. Let $v_{in} \in \texttt{Val}_{in}$ and pc_{exit} be the input value and the exit point of the constructor. The semantic of the class constructor `init`, $S[\![\texttt{init}]\!] \in (\texttt{Store} \times \texttt{Val} \rightarrow \wp(\texttt{Env} \times \texttt{Store}))$, is defined by*

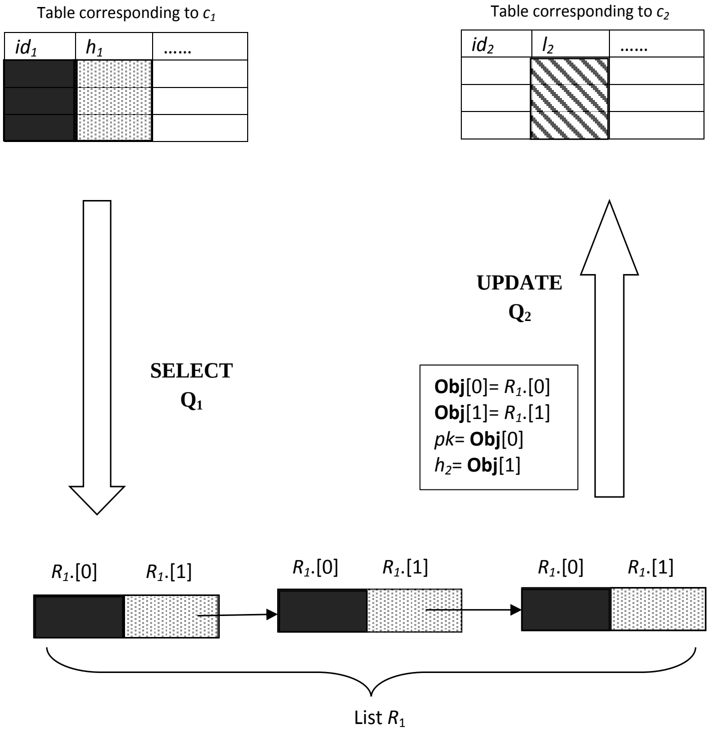
$$S[\![\texttt{init}]\!](s, v_{in}) = \Big\{(e_0, s_0) \mid (e_0 \triangleq V_{in} \rightarrow a_{in}, pc \rightarrow a_{pc}) \wedge (s_0 \triangleq s[a_{in} \rightarrow v_{in}, a_{pc} \rightarrow pc_{exit}])\Big\}$$

Definition 2 (Method Semantics). *Let $\texttt{Val}_{in} \subseteq \texttt{Val}$ and $\texttt{Val}_{out} \subseteq \texttt{Val}$ be the semantic domains for the input values and the output values respectively. Let $v_{in} \in \texttt{Val}_{in}$ be the input values, a_{in} and a_{pc} be the fresh memory locations, and pc_{exit} be the exit point of the method m. The semantic of a method m, $S[\![m]\!] \in (\texttt{Env} \times \texttt{Store} \times \texttt{Val}_{in} \rightarrow \wp(\texttt{Env} \times \texttt{Store} \times \texttt{Val}_{out})$, is defined as*

$$\begin{aligned} S[\![m]\!](e, s, v_{in}) = \Big\{(e', s', v_{out}) \mid (e' \triangleq e[V_{in} \rightarrow a_{in}, pc \rightarrow a_{pc}]) \wedge \\ (s' \triangleq s[a_{in} \rightarrow v_{in}, a_{pc} \rightarrow pc_{exit}]) \wedge v_{out} \in \texttt{Val}_{out}\Big\} \end{aligned}$$

```
Session session = getSessionFactory().openSession();
Transaction tx =session.beginTransaction();
Query $Q_1$ = session.createQuery("SELECT $id_1$, $h_1$ FROM $c_1$");
List $R_1$ = $Q_1$.list();
for(Object[] obj:$R_1$){
 $pk$=(Int) obj[0];
 $h_2$=(Int) obj[1];
 Query $Q_2$ = session.createQuery("UPDATE $c_2$ SET $l_2$ = $l_2$ +1
                 WHERE $id_2$ = $pk$ AND $h_2$=1000");
 int result = $Q_2$.executeUpdate();}
tx.commit();
session.close();
```

(a) A HQL program *P*



(b) Execution view of *P*

Fig. 1: An example HQL program and its execution view

Object semantics in object-oriented languages is defined in terms of interaction history between the program-context and the object.

Set of Interaction States. The set of interaction states in object-oriented languages is defined by

$$\Sigma = \texttt{Env} \times \texttt{Store} \times \texttt{Val}_{out} \times \wp(\texttt{Loc})$$

where $\texttt{Env}$, $\texttt{Store}$, $\texttt{Val}_{out}$, and $\texttt{Loc}$ are the set of application environments, the set of stores, the set of output values, and the set of addresses (escaped ones only) respectively.

Set of Initial Interaction States. The set of initial interaction states is defined by

$$\mathcal{I}_0 = \Big\{\langle e_0, s_0, \phi, \emptyset\rangle \mid S[\![\texttt{init}]\!](v_{in}, s) \ni \langle e_0, s_0\rangle, v_{in} \in \texttt{Val}_{in}\Big\}$$

Observe that ϕ denotes no output produced by the constructor and $\emptyset$ represents the empty context with no escaped address.

Transition Relation. Let $\texttt{Lab} = (\mathbb{M} \times \texttt{Val}_{in}) \cup \{\mathsf{upd}\}$ be a set of labels, where $\mathbb{M}$ is the set of class-methods, $\texttt{Val}_{in}$ is the set of input values and upd denotes an indirect update operation by the context.

The transition relation $\mathscr{T} : \texttt{Lab} \times \Sigma \longrightarrow \wp(\Sigma)$ specifies which successor interaction states $\sigma' = \langle e', s', v', \texttt{Esc}'\rangle \in \Sigma$ can follow (*i*) when an object's methods $m \in \mathbb{M}$ with input $v_{in} \in \texttt{Val}_{in}$ is directly invoked on an interaction state $\sigma = \langle e, s, v, \texttt{Esc}\rangle$ (**direct interaction**), or (*ii*) the context indirectly updates an address escaped from an object's scope (**indirect interaction**).

Definition 3 (Direct Interaction $\mathscr{T}_{dir}$). *Transition on Direct Interaction is defined below:*

$$\mathscr{T}_{dir}[\![(m, v_{in})]\!](\langle e, s, v, \mathit{Esc}\rangle) = \Big\{\langle e', s', v', \mathit{Esc}'\rangle \mid S[\![m]\!](\langle e, s, v_{in}\rangle) \ni \langle e', s', v'\rangle \\ \wedge\ \mathit{Esc}' = \mathit{Esc} \cup \texttt{reach}(v', s')\Big\}$$

where

$$\texttt{reach}(v, s) = \begin{cases} \textit{if } v \in \mathit{Loc} \\ \qquad \{v\} \cup \{\texttt{reach}(e'(f), s) \mid \exists B.\ B = \{\texttt{init}, F, M\}, f \in F, \\ \qquad s(v) \textit{ is an instance of } B,\ s(s(v)) = e' \\ \\ \textit{else } \emptyset \end{cases}$$

Definition 4 (Indirect Interaction $\mathscr{T}_{ind}$). *Transition on Indirect Interaction is defined below:*

$$\mathscr{T}_{ind}[\![\mathsf{upd}]\!](\langle e, s, v, \mathit{Esc}\rangle) = \Big\{\langle e, s', v, \mathit{Esc}\rangle \mid \exists a \in \mathit{Esc}.\ \texttt{Update}(a, s) \ni s'\Big\}$$

where $\texttt{Update}(a, s) = \{s' \mid \exists v \in \mathit{Val}.\ s' = s[a \leftarrow v]\}$

Definition 5 (Transition relation $\mathscr{T}$). *Let $\sigma \in \Sigma$ be an interaction state. The transition relation $\mathscr{T} : \texttt{Lab} \times \Sigma \longrightarrow \wp(\Sigma)$ is defined as $\mathscr{T} = \mathscr{T}_{dir} \cup \mathscr{T}_{ind}$, where $\mathscr{T}_{dir}$ and $\mathscr{T}_{ind}$ represent direct and indirect transitions respectively.*

Concrete Semantics of Session Objects An attractive feature of HQL is the presence of `Hibernate Session` which provides a central interface between the application and Hibernate and acts as persistence manager. A transient object is converted into persistent state when associated with Hibernate `Session`, which has a representation in the underlying database. Various methods in Hibernate `Session` are used to propagate object's states from memory to the database (or vice versa).

We denote the abstract syntax of a `Session` method by a triplet $\langle \mathtt{C}, \phi, \mathtt{OP} \rangle$, where `OP` is the operation to be performed on the database tuples corresponding to a set of objects of classes $c \in \mathtt{C}$ satisfying the condition ϕ. This is depicted in Table 1.

Following [7], the abstract syntax of any SQL statement Q is denoted by a tuple $\langle A, \phi \rangle$, meaning that Q first identifies an active data set from the database using a pre-condition ϕ that follows first-order logic, and then performs the appropriate operations A on the selected data set. For instance, the query "$\mathtt{SELECT}\ a_1,\ a_2\ \mathtt{FROM}\ t\ \mathtt{WHERE}\ a_3 \leq 30$" is denoted by $\langle A, \phi \rangle$ where A represents the action-part "$\mathtt{SELECT}\ a_1,\ a_2\ \mathtt{FROM}\ t$" and ϕ represents the conditional-part "$a_3 \leq 30$". The database environment ρ_d and the table environment ρ_t are defined as [7]:

Database Environment. We consider a database as a set of indexed tables $\{t_i \mid i \in I_x\}$ for a given set of indexes I_x. We define database environment by a function ρ_d whose domain is I_x, such that for $i \in I_x,\ \rho_d(i) = t_i$.

Table Environment. Given a database environment ρ_d and a table $t \in d$. We define $attr(t) = \{a_1, a_2, ..., a_k\}$. So, $t \subseteq D_1 \times D_2 \times .... \times D_k$ where, a_i is the attribute corresponding to the typed domain D_i . A table environment ρ_t for a table t is defined as a function such that for any attribute $a_i \in attr(t)$,

$$\rho_t(a_i) = \langle \pi_i(l_j) \mid l_j \in t \rangle$$

Where π is the projection operator, *i.e.* $\pi_i(l_j)$ is the i^{th} element of the l_j-th row. In other words, ρ_t maps a_i to the ordered set of values over the rows of the table t.

Given a HQL environment $e \in \mathtt{Env}$, a HQL store $s \in \mathtt{Store}$, and a database environment $\rho_d \in \mathfrak{E}_d$. The concrete semantics of `Session` methods are defined by specifying how they are executed on (e, s, ρ_d), resulting into new state $(e', s', \rho_{d'})$. These make the use of the semantics of database statements `SELECT, INSERT, UPDATE, DELETE` [7].

Fix-point Semantics of HQL We extend the notion of interaction states of OOP [11] to the case of HQL, considering the interaction of context with `Session` objects. To this aim, we include database environment in the definition of HQL states. The set of interaction states of HQL is, thus, defined by

$$\Sigma = \mathtt{Env} \times \mathtt{Store} \times \mathfrak{E}_d \times \mathtt{Val}_{out} \times \wp(\mathtt{Loc})$$

where `Env`, `Store`, $\mathfrak{E}_d$, $\mathtt{Val}_{out}$, and `Loc` are the set of application environments, the set of stores, the set of database environments, the set of output values, and the set of addresses respectively.

Constants and Variables			
n	$\in$	$\mathbb{N}$	Set of Integers
v	$\in$	$\mathbb{V}$	Set of Variables
Arithmetic and Boolean Expressions			
exp	$\in$	$\mathbb{E}$	Set of Arithmetic Expressions
exp	::=	$n \mid v \mid exp_1 \oplus exp_2$	
		where $\oplus \in \{+, -, *, /\}$	
b	$\in$	$\mathbb{B}$	Set of Boolean Expressions
b	::=	true $\mid$ false $\mid exp_1 \otimes exp_2 \mid \neg b \mid b_1 \oslash b_2$	
		where $\otimes \in \{\leq, \geq, ==, >, \neq, \dots\}$ and $\oslash \in \{\vee, \wedge\}$	
Well-formed Formulas			
τ	$\in$	$\mathbb{T}$	Set of Terms
τ	::=	$n \mid v \mid f_n(\tau_1, \tau_2, ..., \tau_n)$	
		where f_n is an n-ary function.	
a_f	$\in$	$\mathbb{A}_f$	Set of Atomic Formulas
a_f	::=	$R_n(\tau_1, \tau_2, ..., \tau_n) \mid \tau_1 == \tau_2$	
		where $R_n(\tau_1, \tau_2, ..., \tau_n) \in \{true, false\}$	
ϕ	$\in$	$\mathbb{W}$	Set of Well-formed Formulas
ϕ	::=	$a_f \mid \neg\phi \mid \phi_1 \oslash \phi_2$	
		where $\oslash \in \{\vee, \wedge\}$	
HQL Functions			
$g(\vec{e})$	::=	$\texttt{GROUP BY}(\vec{exp}) \mid id$	
		where $\vec{exp} = \langle exp_1, ..., exp_n \mid exp_i \in \mathbb{E} \rangle$	
r	::=	`DISTINCT` $\mid$ `ALL`	
s	::=	`AVG` $\mid$ `SUM` $\mid$ `MAX` $\mid$ `MIN` $\mid$ `COUNT`	
$h(exp)$	::=	$s \circ r(exp) \mid \texttt{DISTINCT}(exp) \mid id$	
$h(*)$	::=	`COUNT(*)`	
		where * represents a list of database attributes denoted by $\vec{v_d}$	
$\vec{h}(\vec{x})$	::=	$\langle h_1(x_1), ..., h_n(x_n) \rangle$	
		where $\vec{h} = \langle h_1, ..., h_n \rangle$ and $\vec{x} = \langle x_1, ..., x_n \mid x_i = exp \vee x_i = * \rangle$	
$f(\vec{exp})$	::=	$\texttt{ORDER BY ASC}(\vec{exp}) \mid \texttt{ORDER BY DESC}(\vec{exp}) \mid id$	
Session Methods			
c	$\in$	`Class`	Set of Classes
c	::=	$\langle \texttt{init}, \texttt{F}, \texttt{M} \rangle$	
		where `init` is the constructor, $\texttt{F} \subseteq \texttt{Var}$ is the set of fields, and `M` is the set of methods.	
m_{ses}	$\in$	$\texttt{M}_{ses}$	Set of `Session` methods
m_{ses}	::=	$\langle \texttt{C}, \phi, \texttt{OP} \rangle$	
		where $\texttt{C} \subseteq \texttt{Class}$	
`OP`	::=	$\texttt{SEL}(f(\vec{exp}'),\ r(\vec{h}(\vec{x})),\ \phi,\ g(\vec{exp}))$	
	$\mid$	$\texttt{UPD}(\vec{v}, \vec{exp})$	
	$\mid$	`SAVE(obj)`	
	$\mid$	`DEL()`	
		where ϕ represents '`HAVING`' clause and `obj` denotes an instance of a class.	

Table 1: Abstract Syntax of `Session` Methods

We now define the transition relation, by considering (*i*) the direct interaction, when a conventional method is directly invoked, (*ii*) the session interaction, when a `Session` method is invoked, and (*iii*) the indirect transition, when context updates any address escaped from the object's scope.

Definition 6 (Transition relation $\mathscr{T}$). *Let $\sigma \in \Sigma$ be an interaction state. The transition relation $\mathscr{T} : \texttt{Lab}\times\Sigma \longrightarrow \wp(\Sigma)$ is defined as $\mathscr{T} = \mathscr{T}_{dir} \cup \mathscr{T}_{ind} \cup \mathscr{T}_{ses}$, where $\mathscr{T}_{dir}$, $\mathscr{T}_{ind}$ and $\mathscr{T}_{ses}$ represent direct, indirect, and session transitions respectively.* `Lab` *represents the set of labels which include* `Session` *methods $\texttt{M}_{ses}$, conventional class methods $\mathbb{M}$, and an indirect update operation* `Upd` *by the context.*

We denote a transition by $\sigma \xrightarrow{a} \sigma'$ when application of a label $a \in \texttt{Lab}$ on interaction state σ results into a new state σ'.

Let $\mathcal{I}_0$ be the set of initial interaction states. The fix-point trace semantics of HQL program P is defined as

$$\mathscr{T}[\![P]\!](\mathcal{I}_0) = \text{lfp}_{\emptyset}^{\subseteq}\mathcal{F}(\mathcal{I}_0) = \bigcup_{i\leq\omega}\mathcal{F}^i(\mathcal{I}_0)$$

where $\mathcal{F}(\mathcal{I}) = \lambda\mathcal{T}.\ \mathcal{I} \cup \Big\{\sigma_0 \xrightarrow{a_0} \ldots \xrightarrow{a_{n-1}} \sigma_n \xrightarrow{a_n} \sigma_{n+1} \mid \sigma_0 \xrightarrow{a_0} \ldots \xrightarrow{a_{n-1}} \sigma_n \in \mathcal{T}$
$\wedge \sigma_n \xrightarrow{a_n} \sigma_{n+1} \in \mathscr{T}\Big\}$

3.2 Abstract Semantics

Authors in [16, 17] used the Abstract Interpretation framework [3, 4] to define an abstract semantics of imperative languages using symbolic domain of positive propositional formulae in the form

$$\bigwedge_{0\leq i\leq n,\ 0\leq j\leq m} \{y_i \longrightarrow z_j\}$$

which means that the values of variable z_j possibly depend on the values of variable y_i. Later, [8] extends this to the case of structured query languages. The computation of abstract semantics of a program in the propositional formulae domain provides a sound approximation of variable dependences, which allows to capture possible information flow in the program. The information leakage analysis is then performed by checking the satisfiability of formulae after assigning truth values to variables based on their sensitivity levels.

An abstract state $\sigma^\sharp \in \Sigma^\sharp \equiv \mathbb{L}\times\texttt{Pos}$ is a pair $\langle \ell, \psi\rangle$ where $\psi \in \texttt{Pos}$ represents the variables dependences, in the form of propositional formulae, among program variables up to the program label $\ell \in \mathbb{L}$.

Methods in HQL include a set of imperative statements[3]. We assume, for the sake of the simplicity, that attackers are able to observe public variables inside of the main method only. Therefore, our analysis only aims at identifying variable dependences at input-output labels of methods.

[3] For a detailed abstract transition semantics of imperative statements, see [16].

The abstract transition semantics of constructors and conventional methods are defined below.

Definition 7 (Abstract Transition Semantics of Constructor). *Consider a class* $c = \langle \texttt{init}, \texttt{F}, \texttt{M} \rangle$ *where* `init` *is a default constructor. Let* ℓ *be the label of* `init`*. The abstract transition semantics of* `init` *is defined as*

$$\mathscr{T}^{\sharp}[\![{}^{\ell}\texttt{init}]\!] = \{(\ell, \psi) \rightarrow (\mathtt{Succ}({}^{\ell}\texttt{init}), \psi)\}$$

where $\mathtt{Succ}({}^{\ell}\texttt{init})$ *denotes the successor label of* `init`*. Observe that the default constructor is used to initialize the objects-fields only, and it does not add any new dependence.*

The abstract transition semantics of parameterized constructors are defined in the same way as of class methods $m \in \texttt{M}$.

Definition 8 (Abstract Transition Semantics of Methods). *Let* $U \in \wp(\mathtt{Var})$ *be the set of variables which are passed as actual parameters when invoked a method* $m \in \texttt{M}$ *on an abstract state* (ℓ, ψ) *at program label* ℓ*. Let* $V \in \wp(\mathtt{Var})$ *be the formal arguments in the definition of* m*. We assume that* $U \cap V = \emptyset$*. Let* a *and* b *be the variable returned by* m *and the variable to which the value returned by* m *is assigned. The abstract transition semantics is defined as*

$$\mathscr{T}^{\sharp}[\![{}^{\ell}m]\!] = \{(\ell, \psi) \rightarrow (\mathtt{Succ}({}^{\ell}m), \psi')\}$$

where $\psi' = \{x_i \longrightarrow y_i \mid x_i \in U, y_i \in V\} \cup \{a \longrightarrow b\} \cup \psi$ *and* $\mathtt{Succ}({}^{\ell}m)$ *is the label of the successor of* m*.*

We classify the high-level HQL variables into two distinct sets: $\mathtt{Var}_d$ and $\mathtt{Var}_a$. The variables which have a correspondence with database attributes belong to the set $\mathtt{Var}_d$. Otherwise, the variables are treated as usual variables and belong to $\mathtt{Var}_a$. We denote variables in $\mathtt{Var}_d$ by the notation $\overline{v}$, in order to differentiate them from the variables in $\mathtt{Var}_a$. This leads to four types of dependences which may arise in HQL programs: $x \longrightarrow y$, $\overline{x} \longrightarrow y$, $x \longrightarrow \overline{y}$ and $\overline{x} \longrightarrow \overline{y}$, where $x, y \in \mathtt{Var}_a$ and $\overline{x}, \overline{y} \in \mathtt{Var}_d$.

The abstract labeled transition semantics of various `Session` methods are defined in Table 2, where by $\mathtt{Var}(exp)$ and $\mathtt{Field}(c)$ we denote the set of variables in exp and the set of class-fields in the class c respectively. The function $\mathtt{Map}(v)$ is defined as:

$$\mathtt{Map}(v) = \begin{cases} \overline{v} \text{ if } v \text{ has correspondence with a database attribute,} \\ \\ v \text{ otherwise.} \end{cases}$$

Notice that in Table 2 the notation $\widetilde{v}$ stands for either v or $\overline{v}$.

Let $\mathtt{SF}(\psi)$ denotes the set of subformulas in ψ, and the operator $\ominus$ is defined by $\psi_1 \ominus \psi_2 = \bigwedge\Big(\mathtt{SF}(\psi_1) \backslash \mathtt{SF}(\psi_2)\Big)$.

$$\mathscr{T}^{\sharp}\llbracket {}^{\ell}m_{save} \rrbracket$$
$$\stackrel{def}{=} \mathscr{T}^{\sharp}\llbracket {}^{\ell}(\mathtt{C}, \phi, \mathtt{SAVE(obj)}) \rrbracket$$
$$\stackrel{def}{=} \mathscr{T}^{\sharp}\llbracket {}^{\ell}(\{c\}, \mathtt{FALSE}, \mathtt{SAVE(obj)}) \rrbracket$$
$$\stackrel{def}{=} \{\langle \ell, \psi \rangle \xrightarrow{\mathtt{SAVE}} \langle \mathtt{Succ}({}^{\ell}m_{save}), \psi \rangle\}$$

$$\mathscr{T}^{\sharp}\llbracket {}^{\ell}m_{upd} \rrbracket$$
$$\stackrel{def}{=} \mathscr{T}^{\sharp}\llbracket {}^{\ell}(\mathtt{C}, \phi, \mathtt{UPD}(\vec{v}, \vec{exp})) \rrbracket$$
$$\stackrel{def}{=} \mathscr{T}^{\sharp}\llbracket {}^{\ell}(\{c\}, \phi, \mathtt{UPD}(\vec{v}, \vec{exp})) \rrbracket$$
$$\stackrel{def}{=} \{\langle \ell, \psi \rangle \xrightarrow{\mathtt{UPD}} \langle \mathtt{Succ}({}^{\ell}m_{upd}), \psi' \rangle\}$$
$$\text{where } \psi' = \bigwedge \left\{\widetilde{y} \rightarrow \overline{z}_i \mid y \in \mathtt{Var}\llbracket \phi \rrbracket, \widetilde{y} = \mathtt{Map}(y),\ \overline{z}_i \in \vec{v}\right\} \bigcup$$
$$\bigwedge \left\{\widetilde{y_i} \rightarrow \overline{z_i} \mid y_i \in \mathtt{Var}\llbracket exp_i \rrbracket, exp_i \in \vec{exp}, \widetilde{y_i} = \mathtt{Map}(y_i), \overline{z}_i \in \vec{v}\right\} \bigcup \psi''$$
$$\text{and } \psi'' = \begin{cases} \psi \ominus \left(\widetilde{a} \rightarrow \overline{z}_i \mid \overline{z}_i \in \vec{v} \wedge a \in \mathtt{Var} \wedge \widetilde{a} = \mathtt{Map}(a)\right) \text{ if } \phi \text{ is } \mathtt{TRUE} \text{ by default.} \\ \psi \text{ otherwise} \end{cases}$$

$$\mathscr{T}^{\sharp}\llbracket {}^{\ell}m_{del} \rrbracket$$
$$\stackrel{def}{=} \mathscr{T}^{\sharp}\llbracket {}^{\ell}(\mathtt{C}, \phi, \mathtt{DEL}()) \rrbracket$$
$$\stackrel{def}{=} \mathscr{T}^{\sharp}\llbracket {}^{\ell}(\{c\}, \phi, \mathtt{DEL}()) \rrbracket$$
$$\stackrel{def}{=} \{\langle \ell, \psi \rangle \xrightarrow{\mathtt{DEL}} \langle \mathtt{Succ}({}^{\ell}m_{del}), \psi' \rangle\}$$
$$\text{where } \psi' = \bigwedge \left\{\widetilde{y} \rightarrow \overline{z} \mid y \in \mathtt{Var}\llbracket \phi \rrbracket, \widetilde{y} = \mathtt{Map}(y), \overline{z} \in \mathtt{Field}(c)\right\} \bigcup \psi''$$
$$\text{and } \psi'' = \begin{cases} \psi \ominus \left(\widetilde{a} \rightarrow \overline{z}_i \mid \overline{z}_i \in \vec{v} \wedge a \in \mathtt{Var} \wedge \widetilde{a} = \mathtt{Map}(a)\right) \text{ if } \phi \text{ is } \mathtt{TRUE} \text{ by default.} \\ \psi \text{ otherwise} \end{cases}$$

$$\mathscr{T}^{\sharp}\llbracket {}^{\ell}m_{sel} \rrbracket$$
$$\stackrel{def}{=} \mathscr{T}^{\sharp}\llbracket {}^{\ell}(\mathtt{C}, \phi, \mathtt{SEL}(f(\vec{exp}'),\ r(\vec{h}(\vec{x})),\ \phi,\ g(\vec{exp}))) \rrbracket$$
$$\stackrel{def}{=} \{\langle \ell, \psi \rangle \xrightarrow{\mathtt{SEL}} \langle \mathtt{Succ}({}^{\ell}m_{sel}), \psi' \rangle\}$$
$$\text{where } \psi' = \bigwedge \left\{\widetilde{y} \rightarrow \widetilde{z} \mid y \in (\mathtt{Var}\llbracket \phi \rrbracket \cup \mathtt{Var}\llbracket \vec{e} \rrbracket \cup \mathtt{Var}\llbracket \phi' \rrbracket \cup \mathtt{Var}\llbracket \vec{e'} \rrbracket), z \in \mathtt{Var}\llbracket \vec{x} \rrbracket,\right.$$
$$\left.\widetilde{y} = \mathtt{Map}(y), \widetilde{z} = \mathtt{Map}(z)\right\} \bigcup \psi$$

Table 2: Definition of Abstract Transition Function $\mathscr{T}^{\sharp}$ for `Session` methods

4 Information Leakage Analysis

We are now in position to use the abstract semantics defined in the previous section to identify possible sensitive database information leakage through high-level HQL variables. After obtaining over-approximation of variable dependences at each program points, we assign truth values to each variable based on their sensitivity level, and we check the satisfiability of propositional formulae representing variable dependences [16].

Since our main objective is to identify the leakage of sensitive database information possibly due to the interaction of high-level variables, we assume, according to the policy, that different security classes are assigned to database attributes. Accordingly, we assign security levels to the variables in $\texttt{Var}_d$ based on the correspondences. Similarly, we assign the security levels of the variables in $\texttt{Var}_a$ based on their use in the program. For instance, the variables which are used on output channels, are considered as public variables. Observe that for the variables with unknown security class, it may be computed based on the dependence of it on the other application variables or database attributes of known security classes.

Let $\Gamma : \texttt{Var} \rightarrow \{L, H, N\}$ be a function that assigns to each of the variables a security class, either public (L) or private (H) or unknown (N).

After computing abstract semantics of HQL program P, the security class of variables with unknown level (N) in P are upgraded to either H or L, according to the following function:

$$\texttt{Upgrade}(v) = Z \text{ if } \exists\, (u \rightarrow v) \in \mathscr{T}^{\sharp}[\![P]\!].\, \Gamma(u) = Z \wedge \Gamma(u) \neq N \wedge \Gamma(v) = N \qquad (1)$$

We say that program P respects the confidentiality property of database information, if and only if there is no information flow from private to public attributes. To verify this property, a corresponding truth assignment function $\overline{\Gamma}$ is used:

$$\overline{\Gamma}(x) = \begin{cases} T \text{ if } \Gamma(x) = H \\ F \text{ if } \Gamma(x) = L \end{cases}$$

If $\overline{\Gamma}$ does not satisfy any propositional formula in ψ of an abstract state, the analysis will report a possible information leakage.

Let us illustrate this on the running example program P in section 2. According to the policy, let the database attribute corresponding to variable h_1 is private, whereas the attributes corresponding to id_1, id_2 and l_2 are public. Therefore,

$$\Gamma(\overline{h_1}) = H \text{ and } \Gamma(\overline{id_1}) = \Gamma(\overline{id_2}) = \Gamma(\overline{l_2}) = L$$

For other variables in the program, the security levels are unknown. That is,

$$\Gamma(R_1.[0]) = \Gamma(R_1.[1]) = \Gamma(\texttt{obj}[0]) = \Gamma(\texttt{obj}[1]) = \Gamma(pk) = \Gamma(h_2) = N$$

Considering the domain of positive propositional formulae, the abstract semantics yields the following formulae at program point 9 in P:

$$\begin{array}{lllll} \overline{id_1} \rightarrow R_1.[0]; & \overline{h_1} \rightarrow R_1.[1]; & R_1.[0] \rightarrow \texttt{obj}[0]; & R_1.[1] \rightarrow \texttt{obj}[1]; & \\ \texttt{obj}[0] \rightarrow pk; & \texttt{obj}[1] \rightarrow h_2; & pk \rightarrow \overline{l_2}; & \overline{id_2} \rightarrow \overline{l_2}; & h_2 \rightarrow \overline{l_2}; \end{array}$$

According to equation 1, the security levels of the variables with unknown security level N are upgraded as below:

$$\Gamma(R_1.[0]) = L, \Gamma(R_1.[1]) = H, \Gamma(\texttt{obj}[0]) = L, \Gamma(\texttt{obj}[1]) = H$$
$$\Gamma(pk) = L, \qquad \Gamma(h_2) = H$$

Finally, we apply the truth assignment function $\overline{\Gamma}$ which does not satisfy the formula $h_2 \rightarrow \bar{l}_2$, as $\overline{\Gamma}(h_2) = T$ and $\overline{\Gamma}(\bar{l}_2) = F$ and $T \rightarrow F$ is false.

Therefore, the analysis reports that the example program P is vulnerable to leakage of confidential database data.

5 Conclusions

Our approach can capture information leakage on "permanent" data stored in a database when a HQL program manipulates them. This may also lead to a refinement of the non-interference definition that focusses on confidentiality of the data instead of variables. We are now investigating a possible enhancement of the analysis integrating with other abstract domains.

References

1. Bauer, C., King, G.: Hibernate in Action. Manning Publications Co. (2004)
2. Bauer, C., King, G.: Java Persistence with Hibernate. Manning Publications Co. (2006)
3. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 238–252. ACM Press, Los Angeles, CA, USA (1977)
4. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 269–282. ACM Press, San Antonio, Texas (1979)
5. Denning, D.E.: A lattice model of secure information flow. Communications of the ACM 19, 236–243 (1976)
6. Elliott, J., O'Brien, T., Fowler, R.: Harnessing Hibernate. O'Reilly, first edn. (2008)
7. Halder, R., Cortesi, A.: Abstract interpretation of database query languages. Computer Languages, Systems & Structures 38, 123–157 (2012)
8. Halder, R., Zanioli, M., Cortesi, A.: Information leakage analysis of database query languages. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC'14). pp. 813–820. ACM Press, Gyeongju, Korea (24–28 March 2014)
9. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. International Journal of Information Security 8, 399–422 (2009)
10. Li, B.: Analyzing information-flow in java program based on slicing technique. SIGSOFT Software Engineering Notes 27, 98–103 (2002)
11. Logozzo, F.: Class invariants as abstract interpretation of trace semantics. Computer Languages, Systems & Structures 35, 100–142 (2009)
12. Myers, A.C.: Jflow: practical mostly-static information flow control. In: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 228–241. ACM Press, San Antonio, Texas, USA (1999)

13. Pottier, F., Simonet, V.: Information flow inference for ml. ACM Transactions on Programming Languages and Systems 25, 117–158 (2003)
14. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21, 5–19 (2003)
15. Smith, S.F., Thober, M.: Refactoring programs to secure information flows. In: Proceedings of the workshop on Programming languages and analysis for security. pp. 75–84. ACM Press, Canada (2006)
16. Zanioli, M., Cortesi, A.: Information leakage analysis by abstract interpretation. In: Proceedings of the 37th int. conf. on Current trends in theory and practice of computer science. pp. 545–557. Springer LNCS 6543, Nov Smokovec, Slovakia (2011)
17. Zanioli, M., Ferrara, P., Cortesi, A.: Sails: static analysis of information leakage with sample. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC'12). pp. 1308–1313. ACM Press, Trento, Italy (2012)