Data-centric Refinement of Information Flow Analysis of Database Applications

Md. Imran Alam and Raju Halder

Indian Institute of Technology Patna, India
{imran.mtcs13, halder}@iitp.ac.in

Abstract. In the recent age of information, most of the applications are associated with external database states. The confidentiality of sensitive database information may be compromised due to the influence of sensitive attributes on insensitive ones during the computation by database statements. Existing language-based approaches to capture possible leakage of sensitive database information are coarse-grained and are based on the assumption that attackers are able to view all values of insensitive attributes in the database. In this paper, we propose a data-centric approach which covers more generic scenarios where attackers are able to view only a part of the attribute-values according to the policy. This leads to more precise semantic-based analysis which reduces false positives with respect to the literature.

Key words: Information Flow Analysis, Dependence Graph, Database Application

1 Introduction

Protecting private data in computer systems is a promising field of research. While access control and encryption prevent confidential information from being read or modified by unauthorized users, they do not regulate the information propagation after it has been released from the source for execution. Confidentiality may be compromised during the flow of information along the control structure of any software systems [13]. For instance, let l and h be public (or *low*) and private (or *high*) variables respectively, an attacker can guess the sensitive value of h by observing l on the output channel in case of (*i*) assignment statements (*e.g.*, $l := h$) or (*ii*) iteration and conditional statements (*e.g.*, `if` ($h == 0$) `then` $l := 20$; `else` $l := -20$;). The former is called a direct/explicit flow, whereas the later is called indirect/implicit flow. Language-based information flow security analysis [13] has emerged as a promising technique to identify such undesirable information flows in software systems and hence to prevent unauthorized leakage of confidential information.

1.1 Related Works

A series of works on language-based information flow have been proposed for various programming paradigms [15, 10, 8, 12]. The first attempt to prevent

confidential information leakage is based on lattice theoretic model [4] where a partial order is defined among various security labels (*e.g.*, $high \geq low$) and an upward information flow on lattice is allowed to ensure the confidentiality. In [13, 14], authors proposed security type systems considering a set of security types and typing rules which guarantee secure information flow in programs. Type-based systems are not flow-sensitive and may produce false alarm [10]. As an improvement, dependence graph-based approaches [10, 9, 2, 16] are flow-sensitive and they overcome the bottlenecks of security type systems [10]. Static analysis on all possible paths in dependence graphs identifies possible information leakage. An worth-mentioning approach is backward slicing *w.r.t.* *low*-variables [2]. The context-sensitivity and object-sensitivity are considered, as an improvement, in [10]. In practice, dependence graph-based security analysis is limited to realistic program of about 100KLOC [9]. Various formal methods, *e.g.* Abstract Interpretation, Model checking, Axiomatic Rules, etc. [18, 19, 11, 6] are also applied in this direction. The non-interference principle [13] says that a variation of high input must not influence the low view of the applications, and this is the basis of security principle which must be respected by the proposed techniques.

1.2 Motivations

To the best of our knowledge, authors in [8, 3] first proposed information flow analysis to the case of database query languages. The confidentiality of sensitive database information may be compromised due to the influence of sensitive attributes on insensitive ones during computations by database statements. The proposal in [8, 3] uses the abstract interpretation framework to capture attributes dependences at each program points by combining symbolic domain of propositional formula and numerical abstract domain. However, the analysis may produce false alarms when we focus on the dependences based on values instead of attributes. More importantly, the analysis is coarse grained and makes the assumption that an attacker is able to view all values of public attributes.

1.3 Contributions

In this paper, we propose a fine-grained information flow analysis of database applications based on dependence graphs. The proposal covers generic scenarios where attackers are allowed to observe a part of insensitive database information (rather than all) corresponding to public attributes. To this aim, we propose a data-centric computation of dependences in database applications. This leads to a refinement of dependence graphs for database applications, giving rise to a more precise semantics-based analysis of information flow. The main contributions of the paper are:

- We propose a data-centric refinement of Database-Oriented Program Dependency Graph (DOPDG) in order to reduce a number of false data-dependences.
- We perform information flow analysis based on the refined DOPDG to identify possible information leakage in database applications.

The structure of the paper is as follows: In section 2, we discuss in detail the notion of syntax-based DOPDG and its data-centric refinement. Information flow analysis of database applications based on the refined DOPDG is discussed in section 3. The complexity and correctness of our approach is discussed in sections 4. Finally, we conclude our work in section 5.

2 Database-Oriented Program Dependence Graph (DOPDG)

Willmor et al. [17] proposed the notion of dependence graph in case of database applications embedding query languages. This is an extension of traditional Program Dependence Graph (PDG) [7] considering two additional dependences defined below:

Definition 1 (Program-Database (PD) dependence). *A database statement Q is said to be Program-Database dependent on an imperative statement S if it uses a variable x defined by S such that there is no redefinition of x in between S and Q. A dual is also PD-dependence.*

Definition 2 (Database-Database (DD) dependence). *A database statement Q_1 is Database-Database dependent on another database statement Q_2 if Q_1 uses an attribute x which is defined by Q_2 and there is no redefinition of x and no roll-back operation of Q_2 between Q_1 and Q_2.*

The above definitions are syntax-based where dependences depend only on the presence of used and defined variables (either application variables or database attributes) in the statements. To illustrate the construction of syntax-based DOPDG, let us define the following two functions: $\texttt{DEF} : \mathbb{I} \rightarrow \wp(\mathbb{V} \times \texttt{Lab})$ and $\texttt{USE} : \mathbb{I} \rightarrow \wp(\mathbb{V} \times \texttt{Lab})$ where $\mathbb{I}$ is the set of statements (both imperative and database statements), $\mathbb{V}_{db}$ is the set of database attributes, $\mathbb{V}_{app}$ is the set of application variables, $\mathbb{V} = \mathbb{V}_{db} \cup \mathbb{V}_{app}$, $\mathbb{V}_{db} \cap \mathbb{V}_{app} = \emptyset$, and $\texttt{Lab} = \{\texttt{full}, \texttt{partial}\}$. The label "`full`", if associated with database attribute x, denotes that all the values corresponding to the attribute x is defined by database statement, whereas "`partial`" denotes that a subset of the values of the attribute is defined. For instance, an attribute is fully defined when there is no `WHERE` clause in `INSERT`, `DELETE`, `UPDATE` statements. Observe that, in case of application variable, the label is by default always "`full`".

Example 1. Consider the application program `Prog` and the database table `Emp` depicted in Figures 1(a) and 1(b) respectively. The syntax-based DOPDG of `P` is depicted in Figure 1(c). The data-dependences between imperative statements and the control dependences are computed following similar approach as in the case of traditional Program Dependence Graphs. To obtain DD- and PD-dependences, the defined- and used-variables are computed as follows:

$\texttt{DEF}(2) = \{(x, \texttt{full})\}$ $\quad\quad$ $\texttt{DEF}(3) = \{(y, \texttt{full})\}$

$\texttt{DEF}(4) = \{(\text{ssn}, \texttt{full}), (\text{name}, \texttt{full}), (\text{salary}, \texttt{full})\}$

$\texttt{DEF}(5) = \{(\text{salary}, \texttt{partial})\}$ $\quad$ $\texttt{USE}(5) = \{(\text{salary}, \texttt{partial}), (\text{ssn}, \texttt{full}), (x, \texttt{full})\}$

$\texttt{DEF}(6) = \{(\text{salary}, \texttt{partial})\}$ $\quad$ $\texttt{USE}(6) = \{(\text{salary}, \texttt{partial}), (\text{ssn}, \texttt{full}), (y, \texttt{full})\}$

$\texttt{USE}(7) = \{(\text{salary}, \texttt{partial}), (\text{ssn}, \texttt{full})\}$

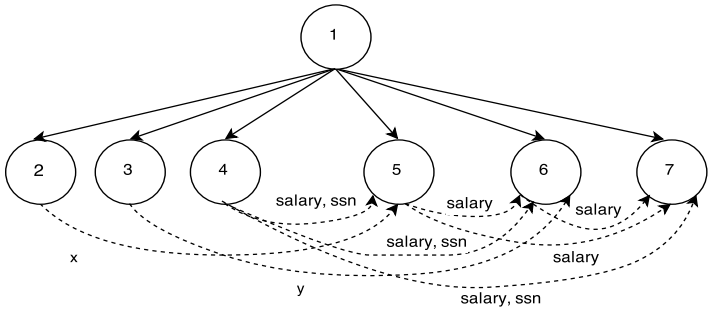
Based on this information, we can easily compute DD- and PD-dependences. For instance, edges $4 \xrightarrow{\text{salary, ssn}} 5$, $5 \xrightarrow{\text{salary}} 6$, etc. represent DD-dependences, whereas edges $2 \xrightarrow{x} 5$, $3 \xrightarrow{y} 6$ represent PD-dependences.

```
1. Start
2. float x = 0.1;
3. float y = 0.2;
4. Statement myStmt = DriverManager.getConnection("jdbc mysql:
        //200.210.220. 1:1114/demo", "scott","tiger").createStatement();
5. UPDATE Emp SET salary = salary + x * salary WHERE ssn BETWEEN 10 AND 30;
6. UPDATE Emp SET salary = salary + y * salary WHERE ssn BETWEEN 35 AND 50;
7. SELECT avg(salary) FROM Emp WHERE ssn BETWEEN 40 AND 90;
8. Stop;
```

(a) Program `Prog`

ssn	name	salary
1	John	20000
2	Smith	14500
3	David	24000
4	Marry	30000
5	Ramu	15000
6	Arun	20000
7	Shayam	16000
8	Lisa	15000
9	Manoj	30000
10	Kumar	24000

(b) Table `Emp`



(c) DOPDG of Program `Prog`

Fig. 1: An example program and its syntax-based DOPDG.

Disadvantage of Syntax-based DOPDGs. This is to be noted that the label "`Lab`" is not enough to remove all false dependences. For instance, in figure 1, although statement 7 is syntactically DD-dependent on statement 5, but semantically there is no dependence because the database-part defined by statement 5 is not used by statement 7. Therefore, we need a more precise semantic-based analysis.

2.1 Condition-Action Rules

In case of database applications, SQL statements define either a part of the values or all of the values corresponding to an attribute depending on the condition present in the `WHERE` clause. This may produce false dependence when the attribute-values defined by one statement does not overlap with the same attribute-values accessed by another statement. The presence of semantic independence, although syntactically dependent, can be identified by considering the Condition-Action rules as suggested by Elena and Jeniffer in [1].

Example 2. Consider the program `Prog` and its syntax-based DOPDG depicted in Figures 1(a) and 1(c) respectively. Analysing dependences between each pair of nodes by applying Condition-Action rule [1], we get that the DD-dependences $5 \xrightarrow{\text{salary}} 6$ and $5 \xrightarrow{\text{salary}} 7$ do not exist. The refined DOPDG is shown in Figure 2.

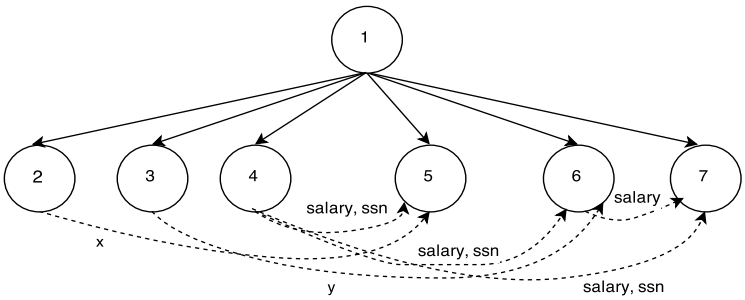


Fig. 2: Refined DOPDG applying Condition-Action rules [1]

Various cases of overlapping of database information defined by Q_2 and subsequently used by Q_1 is depicted using Venn diagram in Figure 3. Dependence occurs due to any of the followings: (*i*) both pre-defined and post-defined values are in the use; (*ii*) pre-defined values are not in the use whereas post-defined values are, (*iii*) pre-defined values are in the use whereas post-defined values are not. This makes the computational complexity of dependence computation exponential *w.r.t.* the number of defining statements.

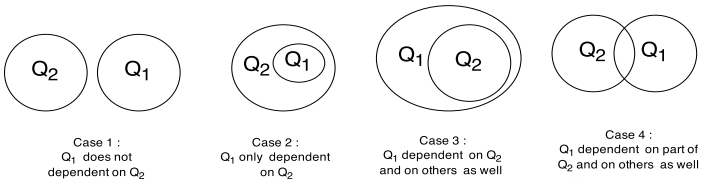


Fig. 3: Overlapping of data defined by Q_2 and used by Q_1

Disadvantage of Condition-Action Rules. The Condition-Action rules [1] can be applied only on a single def-use pair at a time. If more than one database statements (in sequence) partially define an attribute which is then used by another statement, the Condition-Action rules fail to capture semantic independences. Example 3 depicts this.

Example 3. Consider the database application depicted in Figure 4. The DOPDG, applying Condition-Action rules on each pair of def-use (*i.e.* $\ell_1 \xrightarrow{a} \ell_2$, $\ell_1 \xrightarrow{a} \ell_3$, $\ell_1 \xrightarrow{a} \ell_4$, $\ell_2 \xrightarrow{a} \ell_3$, etc.), is depicted in Figure 5(a). However, observe that, since ℓ_4 uses the values within the range 10 to 60 of '*a*', no values defined by statement ℓ_2 can directly affect the observing range of ℓ_4, because ℓ_3 redefines the values. Therefore, the dependence $\ell_2 \xrightarrow{a} \ell_4$ does not exist. A more precise semantics based DOPDG is shown in Figure 5(b).

2.2 Data-centric Refinement of DOPDG

The proposed algorithm is based on data-centric approach to find dependences between database statements. The proposal consists of the following five phases: (*i*) Building of Action Tree, (*ii*) Computing traces, (*iii*) Backwards with precon-

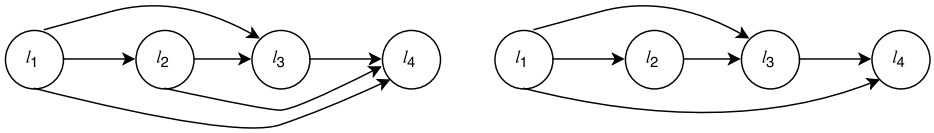
```
ℓ1. Statement myStmt = DriverManager.getConnection("jdbc mysql:
    200.210.220.1:1114/demo", "scott","tiger").createStatement();
ℓ2. UPDATE t SET a = a + 5 WHERE a BETWEEN 20 AND 50;
ℓ3. UPDATE t SET a = a + 1 WHERE a BETWEEN 10 AND 55;
ℓ4. SELECT a FROM t WHERE a BETWEEN 10 AND 60;
```

(a) Program code P

id	*a*	*b*
1	10	5
2	50	9
3	30	15
4	20	40
5	60	30
6	70	10

(b) Table t

Fig. 4: Program P and database table t



(a) Dependences by applying Condition-Action rules [1]

(b) More precise dependences

Fig. 5: Refinement failure by Condition-Action rules

dition, (*iv*) Product of traces and observational-window, and (*v*) Identifying dependences. Let us describe in details each of the phases.

Building of Action Tree. Given a database statement Q, the abstract syntax is denoted by $Q = \langle A, \phi \rangle$ where A and ϕ denote the action-part and condition-part of Q respectively [1]. For instance, the query "`SELECT` b_1, b_2 `FROM` *tab* `WHERE` $b_3 \leq 30$" is denoted by $\langle A, \phi \rangle$ where A represents the action-part "`SELECT` b_1, b_2 `FROM` *tab*" and ϕ represents the conditional-part "$b_3 \leq 30$".

This phase is based on the partitioning of database states using condition-parts present in the defining database statements (*e.g.*, `INSERT`, `DELETE` and `UPDATE`). Given an application containing a set of defining database statements in an order, the partitioning of database information (in the same order) using the condition-parts generates a tree-like structure. We call it *Action Tree*. The nodes of the Action Tree denote actions involved in the defining statements, whereas edges are labeled with the conditions appearing in the condition-parts. The following example illustrates its construction in detail.

Example 4. Let us consider the program code P shown in Figure 4(a). Analyzing the application, the sequence of definitions of attribute '*a*' can be represented in a fixed order: $\ell_1 \rightarrow \ell_2 \rightarrow \ell_3$.

At program point ℓ_1, the statement acts as a defining statement for the values of all attributes in DB. We denote this definition as action part and this is represented by a child node "ℓ_1 : DB" connected with the root node by an edge. As there is no condition-part involved, the edge is labeled by "ℓ_1 : true".

Consider the next defining statement at ℓ_2. The condition-part "$\phi = 20 \leq a \leq 50$" divides the database into two parts: one satisfies ϕ (say, P_ϕ) and other satisfies $\neg\phi$ (say, $P_{\neg\phi}$). The action "$a = a + 5$" is applied on P_ϕ, whereas $P_{\neg\phi}$ remains same. As according to the Condition-Action rules, both $\ell_1 \xrightarrow{P_\phi} \ell_2$ and

$\ell_1 \xrightarrow{P_{\neg\phi}} \ell_2$ exist, we create two children nodes – one denotes the action "$\ell_2 : a' = a + 5$" (the edge labeled by $\ell_2 : \phi$ *i.e.* $\ell_2 : 20 \leq a \leq 50$) and other denotes the action "$\ell_2 : a' = a$ (the edge labeled by $\ell_2 : \neg\phi$ *i.e.* $\ell_2 : a < 20 \wedge a > 50$) [1]. Similar is done for the subsequent statement at ℓ_3 on the result just obtained.

Observe that as $\ell_2 \xrightarrow{P_{\neg\phi}} \ell_3$ (where $\neg\phi = a < 10 \wedge a > 55$) does not exist according to the Condition-Action rules, there is no child node corresponding to the action $a'' = a'$. The resulting Action Tree is depicted in Figure 6.

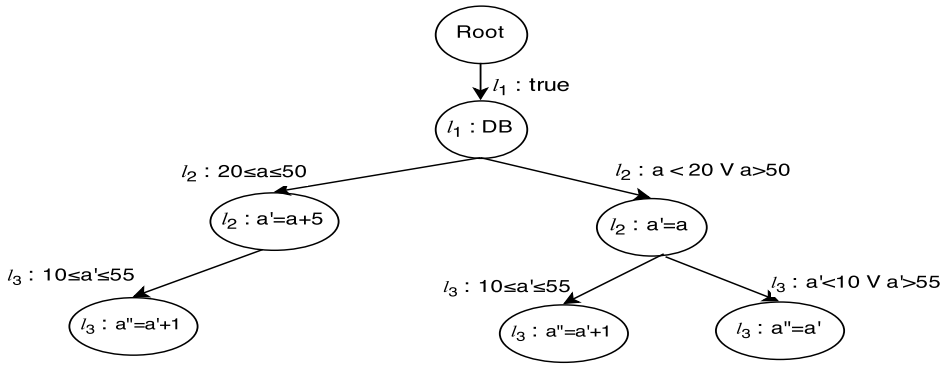


Fig. 6: Action Tree of Program Code P

Computing traces. A trace in an Action Tree is a sequence of labels from the root node to a leaf node. Given an Action Tree, we compute all traces from its root node to all leaves nodes.

Formally, a trace is defined as $\langle\ (\ell_i : \phi,\ \ell_i : A)\ \rangle_{i\geq 1}$ where $\phi \in \mathbb{W}$ (set of well-formed formulas) and $A \in \mathbb{A}$ (set of actions).

Example 5. Given the Action Tree in Figure 6, the traces are:

$\tau_1 = (\ell_1 :\ true,\ \ell_1 :\ DB)(\ell_2 :\ 20 \leq a \leq 50,\ \ell_2 :\ a' = a + 5)(\ell_3 :\ 10 \leq a' \leq 55,\ \ell_3 :\ a'' = a' + 1)$
$\tau_2 = (\ell_1 :\ true,\ \ell_1 :\ DB)(\ell_2 :\ a < 20 \vee a > 50,\ \ell_2 :\ a' = a)(\ell_3 :\ 10 \leq a' \leq 55,\ \ell_3 :\ a'' = a' + 1)$
$\tau_3 = (\ell_1 :\ true,\ \ell_1 :\ DB)(\ell_2 :\ a < 20 \vee a > 50,\ \ell_2 :\ a' = a)(\ell_3 :\ a' < 10 \vee a' > 55,\ \ell_3 :\ a'' = a')$

Backwards with precondition. Hoare logic [5] is a deductive system whose axioms and rules of inference provide a method of proving statements of the form $\{P\}S\{Q\}$, where S is a program statement and P and Q are assertions about the values of variables. This is known as Hoare triple which means that Q (the "postcondition") holds in any state reached by executing S from an initial state in which P (the "precondition") holds. For instance, $\{j = 3 \wedge k = 4\}j := j + k\{j = 7 \wedge k = 4\}$.

As our objective is to find all semantics-based dependences for the use of an attribute on all previous definitions of it, we assume the condition-part of the used-statement as an observational-window (the viewing range). For instance,

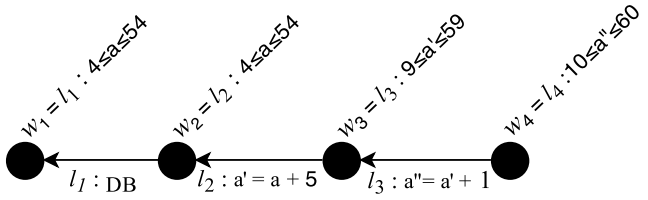
[1] Note that we rename the attributes after an action is performed to distinguish the values before and after the action.

in the running example (Figure 4), the observation window is the condition part of the used-statement at ℓ_4, *i.e.* $10 \leq a'' \leq 60$. Considering the observational-window as postcondition, we apply Hoare logic to compute weakest precondition at each program point appears before. Our aim is to determine the flow of definitions and to verify whether it affects the observational-window. Let us illustrate using running example.

Example 6. Consider the program P shown in Figure 4. The observational-window is $(10 \leq a'' \leq 60)$ at ℓ_4. Let us denote it as $w_4 = \ell_4 : 10 \leq a'' \leq 60$. Applying Hoare logic, we get

$$\{w_1\}\ \ell_1 :\ DB\ \{w_2\}\ \ell_2 :\ a' = a + 5\ \{w_3\}\ \ell_3 :\ a'' = a' + 1\ \{w_4\}$$

where $w_3 = \ell_3 :\ 9 \leq a' \leq 59$, $w_2 = \ell_2 :\ 4 \leq a \leq 54$ and $w_1 = \ell_1 :\ 4 \leq a \leq 54$. This is depicted pictorially below:



The sequence of these assertions forms an observational trace

$$\tau_o = w_1 w_2 w_3 w_4 = (\ell_1 :\ 4 \leq a \leq 54)(\ell_2 :\ 4 \leq a \leq 54)(\ell_3 :\ 9 \leq a' \leq 59)(\ell_4 :\ 10 \leq a'' \leq 60)$$

Formally, an observation trace is defined as $\langle\ (\ell_j : \phi)\ \rangle_{j \geq 1}$ where $\phi \in \mathbb{W}$ (set of well-formed formulas).

Product of action-tree traces and observational trace. Given an action-tree trace τ and an observational trace τ_o, we perform production operation defined as: $\tau \times \tau_o = \langle\ (\ell_i : \phi',\ \ell_i : A')\ \rangle_{i \geq 1} \times \langle (\ell_j : \phi'') \rangle_{j \geq 1} = \langle\ (\ell_x : u,\ \ell_x : v) \rangle_{x \geq 1}$, where

$$(\ell_x : u,\ \ell_x : v) = \begin{cases} \text{if } \exists i, j :\ i = j \text{ and } \phi' \wedge \phi'' \neq \emptyset \\ \qquad (\ell_i : \phi',\ \ell_i : A') \times (\ell_j : \phi'') = (\ell_i : \texttt{true},\ \ell_i : A') \\ \text{if } \exists i, j :\ i = j \text{ and } \phi' \wedge \phi'' = \emptyset \\ \qquad (\ell_i : \phi',\ \ell_i : A') \times (\ell_j : \phi'') = (\ell_i : \texttt{false},\ \ell_i : A') \\ (\ell_j : \phi'',\ \ell_j : \texttt{observe}) \quad \text{if } \nexists i :\ j = i \end{cases}$$

Example 7. Consider the running example. The product of the action-tree traces (τ_1, τ_2, τ_3) and observational trace (τ_o) results the following:

$\tau_1 \times \tau_o$	$(\ell_1 : \texttt{true},\ \ell_1 :\ DB)(\ell_2 : \texttt{true}, \ell_2 :\ a' = a + 5)(\ell_3 : \texttt{true}, \ell_3 :\ a'' = a' + 1)(\ell_4 :\ 10 \leq a'' \leq 60, \ell_4 : \texttt{observe})$
$\tau_2 \times \tau_o$	$(\ell_1 : \texttt{true},\ \ell_1 :\ DB)(\ell_2 : \texttt{true}, \ell_2 :\ a' = a)(\ell_3 : \texttt{true}, \ell_3 :\ a'' = a' + 1)(\ell_4 :\ 10 \leq a'' \leq 60, \ell_4 : \texttt{observe})$
$\tau_3 \times \tau_o$	$(\ell_1 : \texttt{true},\ \ell_1 :\ DB)(\ell_2 : \texttt{true}, \ell_2 :\ a' = a)(\ell_3 : \texttt{true}, \ell_3 :\ a'' = a')(\ell_4 :\ 10 \leq a'' \leq 60, \ell_4 : \texttt{observe})$

Table 1: Product of action-tree traces and observational-window trace.

Identifying dependences. In this phase, a conversion of the traces is performed by masking the actions by either "yes" or "no". An action which may change states is replaced by "yes". Otherwise, it is replaced by "no".

Given a set of masked traces $\mathbb{T}$. We define the following filtering function which eliminates a set of elements from the masked traces which are irrelevant *w.r.t.* the semantics-based dependences:

$$\texttt{Filter}(\tau) = \tau \backslash \{(\ell_k : x, \ell_k : y) \mid x = \texttt{false} \vee y = \texttt{no}\}$$

Given a trace element $e = (\ell_k : x, \ell_k : y)$, suppose $\texttt{Label}(e) = k$ returns the label in e. The following function extracts the semantics-based dependences from the refined traces as follows:

$$\texttt{DEP}(\tau) = \texttt{DEP}(\langle e_1 e_2 e_3 \ldots\ldots e_n \rangle) = (\texttt{Label}(e_{n-1}) \rightarrow \texttt{Label}(e_n))$$

Example 8. In our running example, consider the set of traces obtained after performing product operation in Table 1. The set of masked traces is shown in Table 2(a). The result of applying `Filter` on the masked traces is shown in Table 2(b). Applying the function `DEP` on the filtered masked traces, we get the semantics-based dependences depicted in Table 2(c).

$\tau_1^m = (\ell_1 : \texttt{true}, \ell_1 : \texttt{yes})(\ell_2 : \texttt{true}, \ell_2 : \texttt{yes})(\ell_3 : \texttt{true}, \ell_3 : \texttt{yes})(\ell_4 : 10 \leq a'' \leq 60, \ell_4 : \texttt{observe})$
$\tau_2^m = (\ell_1 : \texttt{true}, \ell_1 : \texttt{yes})(\ell_2 : \texttt{true}, \ell_2 : \texttt{no})(\ell_3 : \texttt{true}, \ell_3 : \texttt{yes})(\ell_4 : 10 \leq a'' \leq 60, \ell_4 : \texttt{observe})$
$\tau_3^m = (\ell_1 : \texttt{true}, \ell_1 : \texttt{yes})(\ell_2 : \texttt{true}, \ell_2 : \texttt{no})(\ell_3 : \texttt{true}, \ell_3 : \texttt{no})(\ell_4 : 10 \leq a'' \leq 60, \ell_4 : \texttt{observe})$

(a) Masked traces

$\texttt{Filter}(\tau_1^m) = (\ell_1 : \texttt{true}, \ell_1 : \texttt{yes})(\ell_2 : \texttt{true}, \ell_2 : \texttt{yes})(\ell_3 : \texttt{true}, \ell_3 : \texttt{yes})(\ell_4 : 10 \leq a'' \leq 60, \ell_4 : \texttt{observe})$
$\texttt{Filter}(\tau_2^m) = (\ell_1 : \texttt{true}, \ell_1 : \texttt{yes})(\ell_3 : \texttt{true}, \ell_3 : \texttt{yes})(\ell_4 : 10 \leq a'' \leq 60, \ell_4 : \texttt{observe})$
$\texttt{Filter}(\tau_3^m) = (\ell_1 : \texttt{true}, \ell_1 : \texttt{yes})(\ell_4 : 10 \leq a'' \leq 60, \ell_4 : \texttt{observe})$

(b) Applying `Filter` on masked traces

$\texttt{DEP}(\texttt{Filter}(\tau_1^m))$	$\ell_3 \rightarrow \ell_4$
$\texttt{DEP}(\texttt{Filter}(\tau_2^m))$	$\ell_3 \rightarrow \ell_4$
$\texttt{DEP}(\texttt{Filter}(\tau_3^m))$	$\ell_1 \rightarrow \ell_4$

(c) Semantics-based dependences

Table 2: Identifying semantics-based dependences

3 Information Flow Security Analysis

In this section, we extend dependence graph-based information flow security analysis to the case of database applications. Given two nodes x and y in a DOPDG, a path $x \xrightarrow{*} y$ denotes that an information flows from x to y. If there is no such path, then there is no information flow.

As first step of the analysis to catch illegal flow of secure information, the attributes and program variables are assigned to various security levels according to their sensitivity. For simplicity, we assume only two security levels: *high* (for private attributes/variables which contain sensitive values) and *low* (for public attributes/variables which contain insensitive values). The finite set of security levels forms a complete lattice with partial order $\leq$ ($x \leq y$ represents x is of lower security level than that of y).

We consider a fine grained scenario where observers are able to see a part of values corresponding to the public attributes. Let us assume that the output statements (imperative or database statements) are considered as hotspots in database applications, *i.e.* the security levels of the variables in output statements are *low*. In order to find the possibility of information leakage, (*i*) we perform a refinement of DOPDG based on the observable range of the public attributes in the output statements, and then (*ii*) we compute a backward slice *w.r.t.* slicing criterion. Slicing criteria include a set of public attributes/variables along with the program point of the output statement. If an attribute or a variable presents in the slice with *high* security level, the analysis reports a possible information leakage.

Example 9. Let us consider the database application `AP` in Figure 7(a) which interacts with a relational database whose schema is `Customer`(ID, CustName, Address, TransAmt, OfferAmt). Suppose the attributes ID, CustName, Address and OfferAmt have *low* security level, whereas TransAmt has *high* security level. The data-centric DOPDG of `AP` is shown in Figure 7(b). The backward slicing of the DOPDG *w.r.t.* slicing criterion $c = \langle 35, \{\text{ID, CustName, OfferAmt}\}\rangle$ produces following slice of the code: $\texttt{BS}(c) = \{4, 5, 6, 7, 11, 15\}$. As the result includes statement 5 which involves *high* variable "TransAmt", this means that there exists a path $5 \xrightarrow{*} 35$ indicating a flow of secure information about "TransAmt" to the output channel. Therefore, the program is not secure.

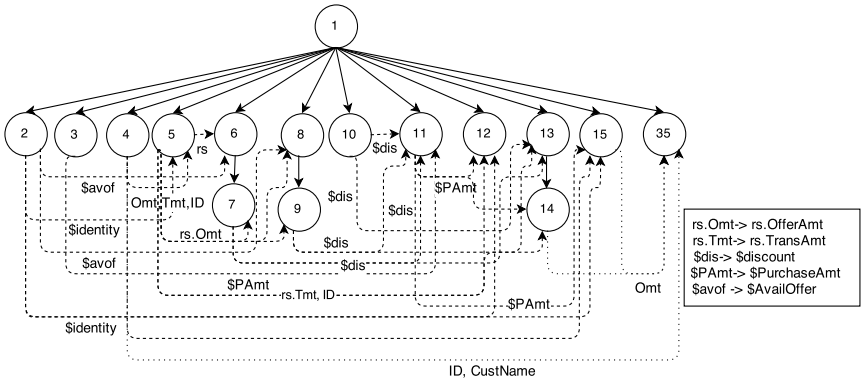
4 Complexity and Correctness

Complexity Analysis. The worst-case time complexity to construct action-tree, assuming n defining statements for an attribute, is $O(2^n)$. Extending this for all m attributes defined by the program, the worst-case time complexity is $O(m \times 2^n)$. Trace generation in the second phase can also be done during the first phase while constructing the action-tree, reducing the extra computational overhead. In the third phase, pre-condition computation based on the Hoare-logic uses Theorem Prover which requires exponential time with respect to the length (say, l) of the conditions. However, we assume that database statements involve simple form of conditions which makes the analysis practically feasible. The time complexity of the remaining phases are linear, $O(n)$. Therefore, the overall worst-case time complexity of the proposed method is $O(m \times 2^n)$, assuming $l \ll n$.

Correctness. Let DB be the database state. Let $D \subseteq DB$ and $U \subseteq DB$ be the *defined* database-part (by statement S_d at program point l) and *used* database-part (by statement S_u at program point k) respectively. The correctness of the proposed method states that if $D \cap U = \emptyset$ then there is no dependence between S_d and S_u *i.e.* $l \nrightarrow k$. We prove this by contrapositive. Let us assume that the proposed method determines a dependence between S_d and S_u for attribute x, *i.e.*, $l \xrightarrow{x} k$. This means that our method results a trace $t = e_1e_2 \ldots e_{n-1}e_n$ such that $\texttt{DEP}(t) = \texttt{DEP}(e_1e_2e_3 \ldots\ldots e_{n-1}e_n) = \texttt{Label}(e_{n-1}) \rightarrow \texttt{Label}(e_n) = l \rightarrow k$ where $e_{n-1} = (l : \texttt{true}, l : \texttt{yes})$, $e_n = (k : \phi, k : \texttt{observe})$ and ϕ is the observational-window. If we move backward along the phases of the proposed method, "`true`"

```
1.  Start;
2.  $Identity = input( ), $AvailOffer=input( );
3.  $PurchaseAmt = getPurchase( );
4.  Statement myStmt=DriverManager.getConnection("jdbc mysql://. . . /demo","X","Y").createStatement();
5.  ResultSet rs=myStmt.executeQuery("SELECT TransAmt, OfferAmt FROM Customer WHERE ID =$identity");
6.  if( $AvailOffer == "yes" and rs.OfferAmt > 200 and rs.TransAmt > 10000)
7.          $discount = rs.OfferAmt * 0.02;
8.  else if ( $AvailOffer == "yes" and rs.OfferAmt > 200);
9.          $discount = rs.OfferAmt * 0.01;
10. else $discount = 0;
11. $PurchaseAmt = $PurchaseAmt − $discount;
12. UPDATE Customer SET TransAmt = rs.TransAmt + $PurchaseAmt WHERE ID = $Identity;
13. if ( $discount = = 0)
14.         UPDATE Customer SET OfferAmt = rs.OfferAmt + $PurchaseAmt WHERE ID=$Identity;
15. else UPDATE Customer SET OfferAmt = $PurchaseAmt WHERE ID = $Identity;
        ------------------------
        ------------------------
35. SELECT ID, CustName WHERE OfferAmt > 150 AND OfferAmt < 200;
36. Stop;
```

(a) Application Program AP



(b) DOPDG of Program AP

Fig. 7: Information flow analysis on DOPDG of AP.

and "yes" in e_{n-1} indicates the following: $\exists\ t_a \in$ action-tree traces and $\exists t_o \in$ observational-trace such that $\pi_l(t_a) \times \pi_l(t_o) = (\phi_a, A_a) \times (\phi_o) = (l :$ true, $l : A_a)$ where π_i is the projection operation of i^{th} element from the trace, and $\phi_a \wedge \phi_o \neq \emptyset$. For $j > l$, $\pi_j(t_a) \times \pi_l(t_o) = (x, y)$ where either $x =$ "false" or $y =$ "no" or both, meaning that there is no re-definition of the part which is defined by S_d at l and is used later by S_u at k. Since $\phi_a \wedge \phi_o \neq \emptyset$, therefore $D \cap U \neq \emptyset$.

5 Conclusions and Future Plans

Computation of dependences in database applications is challenging than imperative programs, as variables consider a set of values instead of a single value. Dependence-graph based information flow analysis is a promising technique as it is flow-, context- and object-sensitive. As an extension to database applications, traditional DOPDG may act as flow-insensitive for database information and may give false alarm. Our proposed method is data-centric which reduces false dependences, leading to a more precise semantics-based analysis and covering more generic fine grained scenarios. Although the computational complexity is exponential, this can be reduced significantly by considering Binary Decision Diagram (BDD) as most of the nodes and edge-levels are repeated

in the action-tree. This is our future aim. As the method is based on state space partitioning, this is comparable to the state explosion problem as in the case of model-checking. We are also investigating the possible use of abstraction to cope with such problems.

References

1. Baralis, E., Widom, J.: An algebraic approach to rule analysis in expert database systems (1994)
2. Cavadini, S.: Secure slices of insecure programs. In: Proc. of the ACM symposium on Information, computer and communications security. pp. 112–122. ACM Press, Tokyo, Japan (2008)
3. Cortesi, A., Halder, R.: Information-flow analysis of hibernate query language. In: Proc. of FDSE 2014. pp. 262–274. Springer LNCS 8860, Vietnam (2014)
4. Denning, D.E.: A lattice model of secure information flow. Communications of the ACM 19(5), 236–243 (1976)
5. Dijkstra, E.W., Dijkstra: A discipline of programming, vol. 1. prentice-hall Englewood Cliffs (1976)
6. Dimitrova, R., Finkbeiner, B., Kovács, M., Rabe, M.N., Seidl, H.: Model checking information flow in reactive systems. In: Verification, Model Checking, and Abstract Interpretation. pp. 169–185. Springer (2012)
7. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems (TOPLAS) 9(3), 319–349 (1987)
8. Halder, R., Zanioli, M., Cortesi, A.: Information leakage analysis of database query languages. In: Proceedings of SAC 2014. pp. 813–820. ACM (2014)
9. Hammer, C., Krinke, J., Snelting, G.: Information flow control for java based on path conditions in dependence graphs. In: IEEE International Symposium on Secure Software Engineering. pp. 87–96 (2006)
10. 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(6), 399–422 (2009)
11. Joshi, R., Leino, K.R.M.: A semantic approach to secure information flow. Science of Computer Programming 37(1), 113–138 (2000)
12. Pottier, F., Simonet, V.: Information flow inference for ml. ACM Transactions on Programming Languages and Systems 25, 117–158 (2003)
13. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (2003)
14. Smith, G.: Principles of secure information flow analysis. In: Malware Detection, pp. 291–307. Springer (2007)
15. Smith, G., Volpano, D.: Secure information flow in a multi-threaded imperative language. In: Proceedings of the POPL 98. pp. 355–364. ACM (1998)
16. Taghdiri, M., Snelting, G., Sinz, C.: Information flow analysis via path condition refinement. In: Formal Aspects of Security and Trust, pp. 65–79. Springer (2011)
17. Willmor, D., Embury, S.M., Shao, J.: Program slicing in the presence of database state. In: Proc. of the 20th IEEE International Conference on Software Maintenance. pp. 448–452. IEEE (2004)
18. Zanioli, M., Cortesi, A.: Information leakage analysis by abstract interpretation. In: Proc. of SOFSEM 2011, pp. 545–557. Springer LNCS (2011)
19. Zanioli, M., Ferrara, P., Cortesi, A.: Sails: static analysis of information leakage with sample. In: Proc. of tthe SAC 2012. pp. 1308–1313. ACM (2012)