Refining Dependencies for Information Flow Analysis of Database Applications

Md. Imran Alam

Department of Computer Science and Engineering,
Indian Institute of Technology Patna, India
E-mail: imran.pcs16@iitp.ac.in

Raju Halder

Department of Computer Science and Engineering,
Indian Institute of Technology Patna, India
E-mail: halder@iitp.ac.in

Abstract: Preserving confidentiality of sensitive information in any computing system always remains a challenging issue. One such reason is improper coding of softwares which may lead to the disclosure of sensitive information to unauthorised users while propagating along the code during execution. Language-based information flow security analysis has emerged as a promising technique to prove that program's executions do not leak sensitive information to untrusted users. In this paper, we propose information flow analysis of database applications. The main contributions of the paper are: 1) refinement of dependence graphs for database applications by removing false dependencies; 2) information-flow analysis of database applications using refined dependence graph. Our approach covers a more generic scenario where attackers are able to view only a part of the attribute-values according to the policy, and leads to a more precise semantic-based analysis which reduces false positives with respect to the literature.

This paper is a revised and expanded version of a paper entitled 'Data-Centric Refinement of Information Flow Analysis of Database Applications' presented at SSCC '15, Kochi, Kerala, India, 10–13 August 2015.

1 Introduction

Outsourcing database data, as effective cost-saving strategy, to third-party data-providers is a common practice now-a-days for database owners [30]. One of the prime concerns in such scenario is to ensure the confidentiality of sensitive data [22]. Encryption is always a suitable method to protect sensitive data, but it fails when actual data is required by the database applications to process user requests. On the other hand, access control mechanism can protect sensitive data from being accessed by unauthorized users, but this does not guaranty secure propagation of sensitive data across the application once it is released from the database. Furthermore, trusting third-party database applications may be a reason for possible disclosure of sensitive data to unauthorized users. Improper coding may lead to a flow of sensitive data along the control structure of software systems to the output channels. For instance, let l and h be *public* (or *low*) and *private* (or *high*) variables respectively in the code snippet depicted in Figure 1. Attackers can guess sensitive values of the private

Figure 1: Code snippet leaking values of private attributes h_1 and h_2

```
                         .........
                         .........
6.    Connection c = DriverManager.getConnection(...);
7.    Statement s = c.createStatement();
8.    String upd = "UPDATE tab SET l1 = l1 + 1, l2 = h2 + 1 WHERE h1=0";
9.    int i = c.executeUpdate(upd);
10.   String query = "SELECT l1, l2 FROM tab";
11.   ResultSet rs = s.executeQuery(query);
12.   while(rs.next()){
13.           System.out.println(rs.getString(1)+" "+rs.getString(2));}
                         .........
                         .........
```

attributes h_1 and h_2 by observing values of the public attributes l_1 and l_2 respectively on the output channel at program point 13. The flow of h_2 to l_2 (see statement 8) is called explicit/direct-flow, whereas the flow of h_1 to l_1 (see statement 8) is called implicit/indirect-flow [24].

Language-based information flow security analysis [24] has emerged as a promising technique to identify such undesirable information flows in software systems and hence to prevent unauthorized leakage of confidential information. The prevailing basic semantic notion of secure information flow is non-interference which states that private data cannot be inferred by public data or critical computation cannot be affected by outsider. Goguen and Meseguer [11] defined non-interference as follows:

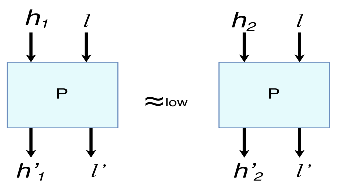
$$\sigma_1 \doteq_{low} \sigma_2 \Longrightarrow [\![P]\!]\sigma_1 \doteq_{low} [\![P]\!]\sigma_2 \tag{1}$$

where P is a program or program statement, σ_1 and σ_2 ($\sigma_1 \neq \sigma_2$) are two states of P before running it, $[\![P]\!]\sigma_1$ and $[\![P]\!]\sigma_2$ are the states obtained after running the program P on σ_1 and σ_2 respectively. $\sigma_1 \doteq_{low} \sigma_2$ implies that two program states are *low*-equivalent if they share same *low* input value. Equation (1) clearly explains that if two input states of a program is *low*-equivalent then executing the program with different values of *high* variables is once again *low*-equivalent. This means that varying *high* input values during execution of a program gives same output. This fact is depicted pictorially in Figure 2. Existing techniques in the literature which are primarily based on Security Type Systems, Program Dependence Graphs, Formal Methods, etc., aim at respecting non-interference principle [24, 1, 9, 32, 15, 29]. While this classical non-interference principle constitutes a reliable guarantee about the flow of information, it is a too restrictive requirement for some real applications demanding declassification of information [25, 20]. For instance, a password-based authentication system violates the non-interference property as its output differs for a given input depending on the stored secret password.

1.1 Motivations and Contributions

To the best of our knowledge, authors in [13, 6] first proposed language-based information flow analysis of database applications embedding SQL. The proposed techniques uses the

Figure 2: Non-Interference Property



abstract interpretation framework to capture attributes dependencies at each program points by combining symbolic domain of propositional formula and numerical abstract domain. However, these techniques suffer from the followings: (*i*) They are coarse-grained and assume that an attacker can see all values of the public attributes, (*ii*) They may introduce false alarms when we focus on the dependencies based on values (semantic) instead of attributes (syntactic), and (*iii*) The choices of suitable abstract domains and best abstractions are challenging under the proposed framework.

In this paper, we propose a fine-grained information flow analysis of database applications based on Database Oriented Program Dependence Graphs (DOPDGs). A DOPDG [31] is an extension of traditional Program Dependence Graph (PDG) [10] considering additional dependencies among imperative and database statements. A Program Dependence Graph (PDG) is a directed graph where nodes represent the program statements and edges represents the data and control dependencies. 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 value-centric computation of dependencies considering non-overlapping of $defined$- and *used*-part of database by various database statements. 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 refinement of Database-Oriented Program Dependency Graph (DOPDG) in order to reduce a number of false data-dependencies in database applications.
- We propose an enhancement of the refinement algorithm using Binary Decision Diagram (BDD).
- We perform information flow analysis based on the refined DOPDG obtained so far to identify possible information leakage in database applications.

The structure of the paper is as follows: In section 2, we survey the current state-of-the-art in the literature. Section 3 recalls the notions of PDG and DOPDG, and we propose some improvements in the construction of DOPDG into more precise one. The value-centric refinement of dependencies in DOPDG is proposed in section 4. An enhancement using Binary Decision Diagram (BDD) to reduce complexity of our algorithm is discussed in section 5. Information flow analysis of database applications based on the refined DOPDG is discussed in section 6. In sections 7, we prove the correctness of our algorithm and discuss the complexity of our approach. Finally, we conclude our work in section 8.

2 Related Works

Language-based information flow security analysis has been heavily studied since the pioneer work of Denning [7]. The first attempt by Denning to prevent confidential information leakage is based on lattice theoretic model where a partial order is defined among various security labels (*e.g.*, *high* $\geq$ *low*) and only an upward information flow on lattice is allowed to ensure the confidentiality. Integrity is the dual of confidentiality.

Security type systems [24, 26, 28] attach security levels on variables, expressions, objects and so on in the source code. A set of typing rules determines what security type should be set to a statement or a program and checks the propagation of information according to the typing rules. The advantage of security-type system is that it is capable to capture insecure explicit flow (as according to the typing rule, expression of *hig*h security level cannot be assigned to variable of *low* security level) and insecure implicit flow (as according to the typing rule, if the condition in conditional/iterative statement is guarded with *high* security level then all statements in the body must be of *high* security level). The security-type systems proposed earlier are not flow-sensitive, and therefore, may produce false alarms [15]. For instance, consider the following code:

```
1.     if((secret % 2) = = 0)
2.             leak = 0;
3.     else leak = 1;
              . . .
              . . .
8.     leak = 5;
9.     Print leak;
```

The type system defined in [24] reports that the code above is vulnerable to possible leakage of sensitive information. However, this is to be observed that the code is correct because value of the *low*-variable ‘leak’ is being killed at statement 8 and there is no output statement before statement 8 – therefore the code satisfies non-interference principle [15]. As an improvement, [17] proposed a flow-sensitive security type system (parameterised by the choice of flow lattice) for simple While-language which allows the type of a variable to float and assigns different security types at different points in the program. The typing rules take into account the flow sensitivity by considering the fact that for any given command c, environment Γ, and security domain pc there is an environment Γ' such that $pc \vdash \Gamma\{c\}\Gamma'$ is derivable.

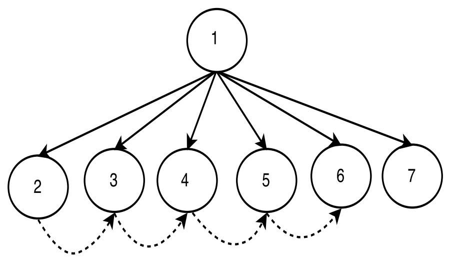
Various forms of dependence graphs (Program Dependence Graph (PDG), System Dependence Graph (SDG), Class Dependence Graph (ClDG), etc. [10, 16, 19, 23]) are immensely useful, as an improvement over type-based systems, for information flow security analysis of programs [15, 14, 5, 27, 29]. In dependence graphs, program statements are represented by nodes whereas data-dependencies and control-dependencies are represented by edges. For instance, Figure 3(b) is a PDG representation of the code shown in Figure 3(a). Static flow-analysis on all possible (data- and control-dependence) paths in the dependence graph identifies possible information leakage in the program. Some of the existing methods include backward slicing, path condition-based analysis, etc. [5, 14]. A backward slice of a program with respect to a program point p and set of program variables V consists of all statements and predicates in the program that may affect the value of variables in V at p. Context-sensitivity in case of programs with procedure-calls, object-sensitivity in case of object-oriented programs are also considered in [15]. In practice, dependence graph-based security analysis is limited to realistic program of about 100KLOC [14]. The main advantage of dependence graph-based approaches is that, although limited in program size, they are flow-, context- and object-sensitive, and can handle realistic programs. However, they suffer from the following drawbacks: First, syntax-based PDG yields false alarms. For example, assume in the code of Figure 3(a) that the security levels of variables x, y are *high* and the security levels of variables z, a are *low*. PDG-based analysis results that the code is insecure as there is a path form node 2 to node 6 indicating a flow of sensitive value defined at node 2 to the output channel at node 6. But this program segment is secure as attackers are unable to judge the values of x and y by observing the value of a on the output channel (because for all possible values of y the value of z remains same *i.e. zero* always). Hence, the code satisfies non-interference principle. Second, although PDG can suitably handle imperative, functional and object-oriented programming languages, however the major challenge lies in case of database statement (UPDATE, INSERT, DELETE) as attribute refers to a list of values rather than single value. Moreover, the presence of trigger, roll-back, commit, etc. in the applications demands a more careful treatment.

Figure 3: An example code snippet and its syntax-based PDG.

(a) Code X

```
1. start;
2. x = getHighInput();
3. y = x + 5;
4. z = 5 * y % 5;
5. a = z + 20;
6. print a;
7. stop;
```

(b) PDG of code X



The problem of insecure information flow has also received much attentions to researchers from formal methods community. Various formal method-based approaches, *e.g.* Abstract Interpretation theory, Hoare Logic, Model Checking, etc., are already proposed to analyze secure information flow in software products [1, 9, 18, 32, 33]. Leino and Joshi [18] introduced a semantics-based approach to analyzing secure information flow based on the semantic equivalence of programs. The proposals in [32, 33] defined the concrete semantics of programs and lift it to an abstract domain suitable for flow analysis. In particular, they consider the domain of propositional formula representing variable's dependencies. The abstract semantics is further refined by combining with numerical abstract domain which improves the precision of the analysis. However, the challenge here is to choose suitable abstract domains and best abstractions. A variety of logical forms are also proposed to characterize information flow security. Amtoft and Banerjee [1] defined prelude semantics by treating program commands as prelude transformer. They introduced a logic based on the Abstract Interpretation of prelude semantics that makes independence between program variables explicit. They used Hoare logic and applied this logic to forward program slicing: forward l-slice is independent of h variables and is secure from information leakage. Authors in [3] defines a set of proof rules for secure information flow based on axiomatic approach. Recently, [9] proposed a model checking-based approach for reactive systems.

3 Database-Oriented Program Dependence Graph (DOPDG)

Dependence Graph is an intermediate representation of programs which makes explicit both the data and control dependencies among program statements [10, 16, 19, 23]. An extension of dependence graph to the case of database applications, known as Database Oriented Program Dependence (DOPDG), is introduced by Willmor et al. [31]. In addition to the traditional data and control dependencies among imperative statements, DOPDG considers additional dependencies defined below:

Program-Database (PD) dependence [31]: A database statement Q is PD-dependent on a statement S if there exists some variable v such that:

- v is defined by S,
- v is used as an input to Q, and
- there is a v-definition free path from S to Q.

Program-Database (PD) dependence [31]: A statement S is PD-dependence on a database statement Q if there exists some variable v such that:

- the execution of Q sets v to be equal to one of the output of Q,
- v is used by S,
- and there is a v-definition free path from Q to S.

Database-Database (DD) dependence [31]: A database statement Q_1 is DD-dependent on another database statement Q_2 iff:

- Q_1.read$\cap$(Q_2.add$\cup Q_2$.upd$\cup Q_2$.del)$\neq \emptyset$,
- there is a roll-back free execution path p between Q_2 and Q_1 (exclusive) such that: Q_2.add$\cap$ p.del$\neq \emptyset$ or Q_2.upd$\cap$ p.upd$\neq \emptyset$ or Q_2.del$\cap$ p.add$\neq \emptyset$.

where Q.add, Q.upd and Q.del represent the part of the database which are inserted, updated and deleted respectively by database statement Q. Observe that the preciseness of the dependence graph depends on how precisely we can identify the overlapping of database-parts by various database operations (add, upd, del).

Let us extend the definitions on dependence graphs from [21] to the case of DOPDG. Table 1 depicts the abstract syntax of database applications defined in [12], where $\vec{e} = \langle e_1, e_2, \ldots, e_m \rangle$ represents a sequence of arithmetic expressions, $\vec{v_d} = \langle v_{d_1}, v_{d_2}, \ldots, v_{d_m} \rangle$ represents a sequence of database attributes, id represents the identity function, and b represents a boolean expression. The notation * represents all database attributes in sequence. A SQL statement Q is denoted by tuple $\langle A, \phi \rangle$ where A represents action and ϕ represents the condition-part (WHERE-Clause) of the statements following first-order formula [12]. Action can be either select or update or insert or delete. The notation ? in the statement $v_a =?$ represents runtime input.

Directed Graph: A directed graph is a pair (N, E) where N is a set of nodes and $E \subseteq N \times N$ is a set of edges. A path p from node n_1 to node n_k is a non-empty sequence of nodes

Table 1 Abstract Syntax of Database Applications embedding SQL

k	$\in$	$\mathbb{K}$ (Constants)
v_d	$\in$	$\mathbb{V}_d$ (Database Attributes)
v_a	$\in$	$\mathbb{V}_a$ (Application Variables)
v	$\in$	$\mathbb{V} = \mathbb{V}_a \cup \mathbb{V}_a$ (All Variables)
ϕ	$\in$	$\mathbb{W}$ (Well-formed Formulas in First Order Logic)
e	$::=$	$k \mid v_a \mid v_d \mid e_1 \oplus e_2 \quad \oplus \in \{+, -, \times, /\}$
b	$::=$	$e_1 == e_2 \mid e_1 \otimes e_2 \quad \otimes \in \{>, \geq, <, \leq, \dots\}$
$g(\vec{e})$	$::=$	GROUP BY$(\vec{e}) \mid id$
r	$::=$	DISTINCT $\mid$ ALL
s	$::=$	AVG $\mid$ SUM $\mid$ MAX $\mid$ MIN $\mid$ COUNT
$h(e)$	$::=$	$s \circ r(e) \mid$ DISTINCT$(e) \mid id$
$h(*)$	$::=$	COUNT(*)
$\vec{h}(\vec{x})$	$::=$	$\langle h_1(x_1), ..., h_n(x_n) \rangle$, where $\vec{h} = \langle h_1, ..., h_n \rangle$ and $\vec{x} = \langle x_1, ..., x_n \rangle$
$f(\vec{e})$	$::=$	ORDER BY ASC$(\vec{e}) \mid$ ORDER BY DESC$(\vec{e}) \mid id$
$\mathbb{Q} \ni Q$	$::=$	$\langle A, \phi \rangle$
A	$::=$	SELECT$(v_a,\ f(\vec{e}),\ r(\vec{h}(\vec{x})),\ \phi,\ g(\vec{e})) \mid$ UPDATE$(\vec{v_d},\ \vec{e}) \mid$ INSERT$(\vec{v_d},\ \vec{e}) \mid$ DELETE$(\vec{v_d})$
$Com \ni c$	$::=$	$skip \mid Q \mid v_a := e \mid v_a :=? \mid output\ e \mid if(b)\ then\ c_1\ else\ c_2\ fi \mid while(b)\ do\ c\ od \mid c_1; c_2$

$\langle n_1, \dots, n_k \rangle \in N^+$ where $(n_i, n_{i+1}) \in E$ for all $i \in \{1, \dots, k-1\}$. A path is called trivial if it is of the form $\langle n \rangle$ (*i.e.*, a sequence of length 1), and non-trivial otherwise. A node n is on the path $\langle n_1, \dots, n_k \rangle$ if $n = n_i$ for some $i \in \{1, \dots, k\}$.

Control Flow Graph: A control flow graph with def and use sets is a tuple (N, E, def, use) where (N, E) is a directed graph, N contains two distinguished nodes start and stop, and def, use: $N \longrightarrow \wp(\mathbb{V})$ are functions returning the defined and used variables set, respectively, for a node.

Program Size: Given $c \in Com$, the number of statements and control conditions in c is denoted by $|c|$ and is defined recursively by: $|skip|$=1, $|Q| = 1$, $|v_a := e|$=1, $|v_a =?| = 1$, $|output\ e| = 1$, $|if(b)\ then\ c_1\ else\ c_2\ fi| = 1 + |c_1| + |c_2|$, and $|while(b)\ do\ c\ od| = 1 + |c|$, $|c_1; c_2| = |c_1|$+$|c_2|$.

Statements and Control Condition Label: For $c \in Com$ and $1 \leq i \leq |c|$ we denote with $c[i]$ the i^{th} statement or control condition in c, which is defined recursively as follows: If $c = skip$ or $c = v_a := e$ or $c = v_a :=?$ or $c = Q$ or $c = output\ e$ then $c[i] = c$ for $1 \leq i \leq |c|$. If $c = if(b)\ then\ c_1\ else\ c_2\ fi$ then $c[1] = e$, $c[i] = c_1[i-1]$ for $1 < i \leq 1 + |c_1|$, and $c[i] = c_2[i - 1 - |c_1|]$ for $1 + |c_1| < i \leq |c|$. If $c = while(b)\ do\ c\ od$ then $c[i] = e$, $c[i] = c_1[i-1]$ for $1 < i \leq |c|$. If $c = c_1; c_2$ then $c[i] = c_1[i]$ for $1 \leq i \leq |c_1|$ and $c[i] = c_2[i - |c_1|]$ for $|c_1| < i \leq |c|$.

Control Flow Graph (CFG): The Control Flow Graph of $c \in Com$ is CFG$_c$ = $(N_c, E_c,$ def$_c$, use$_c$), where the set of nodes is defined as $N_c = \{1, \dots, |c|\} \cup \{start, stop\}$ and the set of edges $E_c \subseteq N_c \times N_c$ is defined recursively by:

- $E_{skip} = E_{v_a:=e} = E_{v_a:=?} = E_Q = E_{output\ e} = \{(start, 1), (1, stop), (start, stop)\}$,
- $E_{if(b)\ then\ c_1\ else\ c_2\ fi} = \{(start, 1), (start, stop)\} \cup$
  $\{(1, n') | (start, n' \ominus 1) \in E_{c_1} \wedge n' \neq stop\} \cup$
  $\{(1, n') | (start, n' \ominus (1 + |c_1|)) \in E_{c_2} \wedge n' \neq stop\} \cup$
  $\{(n, n') | (n \ominus 1, n' \ominus 1) \in E_{c_1} \wedge n' \neq start\} \cup$
  $\{(n, n') | (n \ominus (1 + |c_1|), n' \ominus (1 + |c_1|)) \in E_{c_2} \wedge n' \neq start\}$

- $E_{while(b)\ do\ c\ od} = \{(start, 1), (start, stop)\} \cup$
  $\{(1, n') | (start, n' \ominus 1) \in E_c\} \cup$
  $\{(n', 1) | (n' \ominus 1, stop) \in E_c\} \cup$
  $\{(n, n') | (n \ominus 1, n' \ominus 1) \in E_c \wedge n \neq start \wedge n' \neq stop\}$

- $E_{c_1;c_2} = \{(start, stop)\} \cup$
  $\{(n, n') | (n, n') \in E_{c_1} \wedge n' \neq stop\} \cup$
  $\{(n, n') | (n \ominus |c_1|, n' \ominus |c_1|) \in E_{c_2} \wedge n' \neq start\} \cup$
  $\{(n, n') | (n, stop) \in E_{c_1} \wedge (start, n' \ominus |c_1|) \in E_{c_2} \wedge n \neq start \wedge n' \neq stop\}$

  where the operator $\ominus : (\mathbb{N} \cup \{start, stop\} \times \mathbb{N}) \rightarrow \mathbb{Z} \cup \{start, stop\}$ is defined as:

  $$n \ominus z = \begin{cases} \text{n-z} & \text{if n} \in \mathbb{N} \\ \text{n} & \text{if n} \in \{start, stop\} \end{cases}$$

and the functions def_c, use_c ($\triangleq N_c \rightarrow \wp(\mathbb{V})$) are defined as: for $n \in \{1, \ldots, |c|\}$,

$$\text{def}_c(n) = \begin{cases} \{v_d\} & \text{if } c[n] = v_d := e \text{ or } c[n] = v_d :=? \\ \{v_a\} & \text{if } c[n] = \langle \text{SELECT}(v_a,\ f(\vec{e'}),\ r(\vec{h}(\vec{x})),\ \phi_1,\ g(\vec{e})), \phi \rangle \\ var(\vec{v_d}) & \text{if } c[n] = \langle \text{UPDATE}(\vec{v_d},\ \vec{e}), \phi \rangle \text{ or } c[n] = \langle \text{INSERT}(\vec{v_d},\ \vec{e}), false \rangle \\ & \text{or } c[n] = \langle \text{DELETE}(\vec{v_d}), \phi \rangle \\ \emptyset & \text{otherwise} \end{cases}$$

$$\text{use}_c(n) = \begin{cases} var(e) & \text{if } c[n] = v_d := e \text{ or } c[n] = output\ e \\ var(b) & \text{if } c[n] = b \\ var(\phi) \cup var(\vec{e}) \cup var(\phi_1) \cup var(\vec{x}) \cup var(\vec{e'}) & \text{if } c[n] = \langle \text{SELECT}(v_a,\ f(\vec{e'}),\ r(\vec{h}(\vec{x})), \\ & \quad \phi_1,\ g(\vec{e})), \phi \rangle \\ var(\vec{e}) \cup var(\phi) & \text{if } c[n] = \langle \text{UPDATE}(\vec{v_d},\ \vec{e}), \phi \rangle \\ var(\vec{e}) & \text{if } c[n] = \langle \text{INSERT}(\vec{v_d},\ \vec{e}), false \rangle \\ var(\phi) & \text{if } c[n] = \langle \text{DELETE}(\vec{v_d}), \phi \rangle \\ \emptyset & \text{otherwise} \end{cases}$$

where $var(arg)$ returns a set of variables present in arg.

Data Dependencies: Let $\text{CFG}_c = (N_c, E_c, \text{def}_c, \text{use}_c)$ and $n, n' \in N_c$. If $x \in \text{def}_c(n)$ we say that the definition of x at n reaches n' if there is a non-trivial path p from n to n' such that $x \notin \text{def}_c(n'')$ for every node n'' on p with $n'' \neq n$ and $n'' \neq n'$. Node n' is Data Dependent on node n if there exists $x \in \mathbb{V}$ such that $x \in \text{def}_c(n)$, $x \in \text{use}_c(n')$, and the definition of x at n reaches n'. If ($c[n] = Q$ and $c[n'] \neq Q$) or ($c[n] \neq Q$ and $c[n'] = Q$), then the data dependence is called Program-Database (PD) Dependence. If ($c[n] = c[n'] = Q$), then the data dependence is called Database-Database (PD) Dependence.

Control Dependencies: Let $\text{CFG}_c = (N_c, E_c, \text{def}_c, \text{use}_c)$. Node n' post-dominates node n if $n \neq n'$ and every path from n to $stop$ contains n'. Node n' is control dependent on node n if there is a non-trivial path p from n to n' such that n' post-dominates all nodes $n'' \notin \{n, n'\}$ on p and n' does not post-dominate n.

Database-Oriented Program Dependence Graph (DOPDG): Let $\text{CFG}_c = (N_c, E_c, \text{def}_c, \text{use}_c)$. The directed graph (N', E') is called DOPDG of the CFG if $N' = N_c$ and $(n, n') \in E'$ if and only if n' is data dependent or control dependent on n in CFG.

Example 1: Consider the application program Prog and the database table Emp depicted in Figures 4(a) and 4(b) respectively. The syntax-based DOPDG of Prog is depicted in Figure 4(c). The syntax-based DOPDG consists of a set of nodes corresponding to the

Figure 4: An example program and its syntax-based DOPDG.

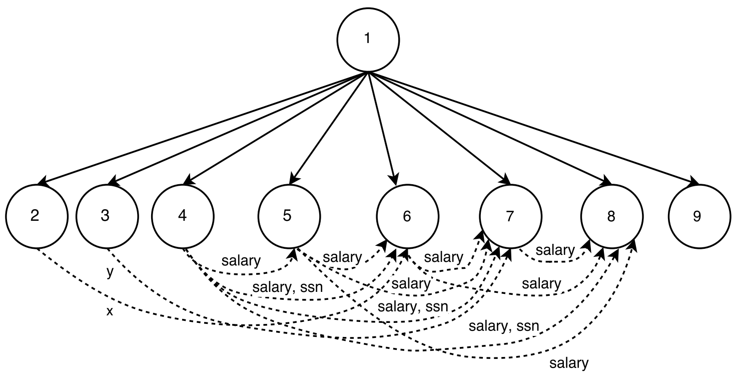
(a) Program Prog

```
1.  start;
2.  float x = 0.1;
3.  float y = 0.2;
4.  Statement s = DriverManager.getConnection("jdbc mysql://200.210.220.
    1:1114/demo", "scott","tiger").createStatement();
5.  int i=s.executeUpdate(UPDATE Emp SET salary = salary + 100);
6.  i=s.executeUpdate(UPDATE Emp SET salary = salary + x * salary WHERE ssn
    BETWEEN 10 AND 30);
7.  i=s.executeUpdate(UPDATE Emp SET salary = salary + y * salary WHERE ssn
    BETWEEN 35 AND 50);
8.  ResultSet r=s.executeQuery(SELECT avg(salary) FROM Emp WHERE ssn
    BETWEEN 40 AND 90);
9.  stop;
```

(b) Table Emp

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

(c) DOPDG of Program Prog



program statements. The data-dependencies between imperative statements and the control dependencies are computed following similar approach as in the case of traditional Program Dependence Graphs. For instance, the edges 1 → 2, 1 → 3, 1 → 4, etc. represent control dependencies. To obtain DD- and PD-dependencies, the defined- and used-variables are computed as follows:

$$\text{def}_c(2) = \{x\}$$
$$\text{def}_c(3) = \{y\}$$
$$\text{def}_c(4) = \{ssn, name, salary\}$$
$$\text{def}_c(5) = \{salary, i\}$$
$$\text{use}_c(5) = \{salary\}$$
$$\text{def}_c(6) = \{salary, i\}$$
$$\text{use}_c(6) = \{salary, ssn, x\}$$

$$\begin{aligned}
\text{def}_c(7) &= \{salary, i\}\\
\text{use}_c(7) &= \{salary, ssn, y\}\\
\text{use}_c(8) &= \{salary, ssn\}
\end{aligned}$$

Based on this information, we can easily compute DD- and PD-dependencies. For instance, edges $4 \rightarrow 5$, $4 \rightarrow 6$, $5 \rightarrow 6$, etc. represent DD-dependencies, whereas edges $2 \rightarrow 6$, $3 \rightarrow 7$ represent PD-dependencies.

3.1 Improvement in Construction of DOPDG using Labels

In this section, we propose an improvement introducing labels along with used-variables, defined-variables to construct more precise DOPDG. The algorithm simply identifies various information (used-variables, defined-variables, whether data is fully/partially-used or fully/partially-defined, etc.) about statements at each program point in the program. Given $c \in Com$ such that $N_c = \{1, \ldots, |c|\} \cup \{start, stop\}$, let us define an improved version of the two functions def_c, use_c:

$$\text{def}_c : N_c \rightarrow \wp(\mathbb{V} \times Lab) \tag{2}$$

$$\text{use}_c : N_c \rightarrow \wp(\mathbb{V} \times Lab) \tag{3}$$

where $\mathbb{V}_d$ is the set of database attributes, $\mathbb{V}_a$ is the set of application variables, $\mathbb{V} = \mathbb{V}_d \cup \mathbb{V}_a$, $\mathbb{V}_d \cap \mathbb{V}_a = \emptyset$, and Lab = {full, partial}. For $n \in \{1, \ldots, |c|\}$,

$$\text{def}_c(n) = \begin{cases}
\{(v_d, \text{full})\} & \text{if } c[n] = v_d := e \text{ or } c[n] = v_d :=?\\
\{(v_a, \text{full})\} & \text{if } c[n] = \langle \text{SELECT}(v_a,\ f(\vec{e'}),\ r(\vec{h}(\vec{x})),\ \phi_1,\ g(\vec{e})), \phi\rangle.\\
var(\vec{v_d}) \times \{\text{full}\} & \text{if } c[n] = \langle \text{UPDATE}(\vec{v_d},\ \vec{e}), \phi\rangle \text{ and } \phi \text{ is tautology.}\\
var(\vec{v_d}) \times \{\text{partial}\} & \text{if } c[n] = \langle \text{UPDATE}(\vec{v_d},\ \vec{e}), \phi\rangle \text{ and } \phi \text{ is not tautology.}\\
var(\vec{v_d}) \times \{\text{partial}\} & \text{if } c[n] = \langle \text{INSERT}(\vec{v_d},\ \vec{e}), false\rangle\\
var(\vec{v_d}) \times \{\text{full}\} & \text{or } c[n] = \langle \text{DELETE}(\vec{v_d}), \phi\rangle \text{ and } \phi \text{ is tautology.}\\
var(\vec{v_d}) \times \{\text{partial}\} & \text{or } c[n] = \langle \text{DELETE}(\vec{v_d}), \phi\rangle \text{ and } \phi \text{ is not tautology.}\\
\emptyset & \text{otherwise}
\end{cases}$$

$$\text{use}_c(n) = \begin{cases}
var(e) \times \{\text{full}\} & \text{if } c[n] = v_d := e \text{ or } c[n] = output\ e\\
var(b) \times \{\text{full}\} & \text{if } c[n] = b\\
((var(\vec{e}) \cup var(\phi_1) \cup var(\vec{x}) \cup var(\vec{e'})) \times \{\text{full}\}) \cup (var(\phi) \times \{\text{full}\}) & \text{if } c[n] = \langle \text{SELECT}(v_a,\ f(\vec{e'}),\ r(\vec{h}(\vec{x})),\ \phi_1,\ g(\vec{e})), \phi\rangle\\
& \text{and } \phi \text{ is tautology.}\\
((var(\vec{e}) \cup var(\phi_1) \cup var(\vec{x}) \cup var(\vec{e'})) \times \{\text{partial}\}) \cup (var(\phi) \times \{\text{full}\}) & \text{if } c[n] = \langle \text{SELECT}(v_a,\ f(\vec{e'}),\ r(\vec{h}(\vec{x})),\ \phi_1,\ g(\vec{e})), \phi\rangle\\
& \text{and } \phi \text{ is not tautology.}\\
(var(\vec{e}) \times \{\text{full}\}) \cup (var(\phi) \times \{\text{full}\}) & \text{if } c[n] = \langle \text{UPDATE}(\vec{v_d},\ \vec{e}), \phi\rangle \text{ and } \phi \text{ is tautology.}\\
(var(\vec{e}) \times \{\text{partial}\}) \cup (var(\phi) \times \{\text{full}\}) & \text{if } c[n] = \langle \text{UPDATE}(\vec{v_d},\ \vec{e}), \phi\rangle \text{ and } \phi \text{ is not tautology.}\\
var(\vec{e}) \times \{\text{full}\} & \text{if } c[n] = \langle \text{INSERT}(\vec{v_d},\ \vec{e}), false\rangle\\
var(\phi) \times \{\text{full}\} & \text{if } c[n] = \langle \text{DELETE}(\vec{v_d}), \phi\rangle\\
\emptyset & \text{otherwise}
\end{cases}$$

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 (*i.e.* ϕ acts as tautology). Observe that, in case of application variable, the label is by default always “full”.

Following examples show how the associated label from “Lab" helps the analysis to identify some false dependencies.

Example 2: Let $c = Q_1;\ Q_2;\ Q_3;\ \ldots$ be a database application where $c[1] = Q_1, c[2] = Q_2, c[3] = Q_3, \ldots$, and suppose that:

$$\begin{aligned}\mathrm{def}_c(1) &= \{(x, \text{partial})\}\\ \mathrm{def}_c(2) &= \{(x, \text{full})\}\\ \mathrm{use}_c(3) &= \{(x, \text{partial})\}\end{aligned}$$

In this case, the label “full” associated with x in $\mathrm{def}_c(2)$ helps to remove the false DD-dependence $1 \xrightarrow{x} 3$, as x is fully redefined by Q_2. Similarly, in the following case where

$$\begin{aligned}\mathrm{def}_c(1) &= \{(x, \text{partial})\}\\ \mathrm{def}_c(2) &= \{(x, \text{partial})\}\\ \mathrm{use}_c(3) &= \{(x, \text{partial})\}\end{aligned}$$

Q_3 is DD-dependent on both Q_1 and Q_2 for x.

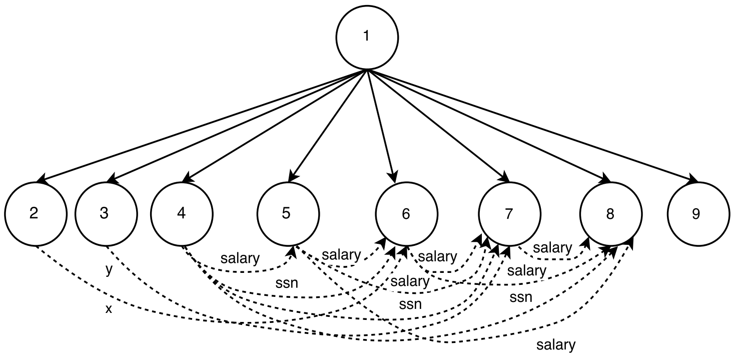
Example 3: Consider the application program Prog and the database table Emp depicted in Figures 4(a) and 4(b) respectively. Following equations 2 and 3, we get:

$$\begin{aligned}\mathrm{def}_c(2) &= \{(x, \text{full})\}\\ \mathrm{def}_c(3) &= \{(y, \text{full})\}\\ \mathrm{def}_c(4) &= \{(ssn, \text{full}), (name, \text{full}), (salary, \text{full})\}\\ \mathrm{def}_c(5) &= \{(salary, \text{full}), (i, \text{full})\}\\ \mathrm{use}_c(5) &= \{(salary, \text{full})\}\\ \mathrm{def}_c(6) &= \{(salary, \text{partial}), (i, \text{full})\}\\ \mathrm{use}_c(6) &= \{(salary, \text{partial}), (ssn, \text{full}), (x, \text{full})\}\\ \mathrm{def}_c(7) &= \{(salary, \text{partial}), (i, \text{full})\}\\ \mathrm{use}_c(7) &= \{(salary, \text{partial}), (ssn, \text{full}), (y, \text{full})\}\\ \mathrm{use}_c(8) &= \{(salary, \text{partial}), (ssn, \text{full})\}\end{aligned}$$

According to this information, we observe that all values of attribute ‘*salary*’ are updated at program point 5, resulting the data-dependencies $4 \xrightarrow{salary} 6$, $4 \xrightarrow{salary} 7$, $4 \xrightarrow{salary} 8$ false. The refined DOPDG is shown in Figure 5.

Limitations: This is to be noted that the label “Lab” is not enough to remove all false dependencies. For instance, in figure 5, although statement 8 is syntactically DD-dependent on 6, but semantically there is no dependence because the database-part defined by statement 6 is not used by statement 8. Therefore, we need a more precise semantics-based analysis.

Figure 5: Refined DOPDG (of Figure 4c) using improved version of def$_c$ and use$_c$ in equations 2 and 3 (data-dependencies $4 \xrightarrow{salary} 6, 4 \xrightarrow{salary} 7, 4 \xrightarrow{salary} 8$ are removed).



3.2 Axiomatic Rules

Expert database systems extend the functionality of conventional database systems by providing a facility for creating and automatically executing Condition-Action rules [4]. A Condition-Action rule in this scenario defined as follows:

$$E_{cond} \longrightarrow E_{act}$$

Where E_{cond} and E_{act} represent the rule's condition as an expression in the extended relational algebra and the rule's action as a data modification operation like INSERT, UPDATE and DELETE respectively. 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 of modification statements. This may produce false dependence when a part defined by database statement Q_2 may not be subsequently used by Q_1. That is, the values defined by Q_2 does not overlap with the values accessed by Q_1.

The presence of semantic independence, although syntactically dependent, can be identified by considering the Condition-Action rules as suggested by Elena and Jeniffer in [4]. Below is an example from [4] which describes the refinement of dependencies based on action-condition rules:

Example 4: Consider a rule X: if an account has a balance less than 500 and an interest rate greater than 0% then that account's interest rate is set to 0%. The rule is expressed as $E^X_{cond} \rightarrow E^X_{upd}$:

$$E^X_{cond} = \Pi_{bal,rate}(\sigma_{bal<500 \wedge rate>0} \text{ACCT})$$

$$E^X_{upd} = \xi[rate' = 0](\sigma_{bal<500 \wedge rate>0} \text{ACCT})$$

Consider rule Y: when the number of customers living in San Francisco exceeds 1000 then the interest rate of all San Francisco customers accounts with a balance greater than 5000 and an interest rate less than 3% is increased by 1%. The rule is expressed as $E^Y_{cond} \rightarrow E^Y_{upd}$ with

$$\mathrm{E}^Y_{cond} = \Pi_{city,C}(\sigma_{C>5000}(A[C = count(name)](\sigma_{city='\mathrm{SF}'}\mathrm{CUST})))$$

$$\mathrm{E}^Y_{upd} = \xi[rate' = rate + 1]\mathrm{E}^Y_c$$

$$\mathrm{E}^Y_c = \sigma_{bal>5000 \wedge rate<3}\mathrm{ACCT} \ltimes_{name} \sigma_{city='\mathrm{SF}'}\mathrm{CUST}$$

Now consider the condition E^X_{cond} in rule X and the update action E^Y_{upd}in rule Y:

$$\begin{aligned} \mathrm{C} =& \mathrm{E}^X_{cond} = \sigma_{bal<500 \wedge rate>0}ACCT \\ \mathrm{A} =& \mathrm{E}^Y_{upd} = \xi[rate' = rate + 1]\mathrm{E}^Y_c \\ \mathrm{E}^Y_c =& (\sigma_{bal>5000 \wedge rate<3}\mathrm{ACCT} \ltimes_{name} \sigma_{city='\mathrm{SF}'}\mathrm{CUST}) \end{aligned}$$

While defining rules, Elena and Jeniffer consider three possibilities: (*i*) both pre-defined and post-defined values by A are in the use by C; (*ii*) pre-defined values by A are not in the use by C whereas post-defined values by A are in use by C, (*iii*) pre-defined values by A are in the use whereas post-defined values by A are not in use by C. This makes the computational complexity of dependence computation exponential *w.r.t.* the number of defining statements.

According to the Action-Condition rules defined in [4], observe that the predicates $bal < 500$ and $bal > 5000$ are contradictory – meaning that the action A operates on a part of data which is not accessed by the condition C. In other words, the action A does not affect the condition C resulting a data-independence.

Example 5: Consider the program Prog and its Refined DOPDG using improved version of def_c and use_c depicted in Figures 4(a) and 5 respectively. Analysing dependencies between each pair of nodes by applying Condition-Action rule [4], we get that the DD-dependencies $6 \xrightarrow{salary} 7$ and $6 \xrightarrow{salary} 8$ do not exist. The refined DOPDG is shown in Figure 6.

Let us now provide a general view of dependence in database applications. Suppose Q_1 is a database statement defining an attribute x and Q is a database statement using x. There is no redefinition of x between Q_1 and Q. The overlapping of the database information defined by Q_1 and subsequently used by Q is depicted using Venn diagram shown in the figure 7, described below:

Case 1: This shows that Q_1 modifies a database-part which is not accessed by Q at all. Therefore, there is no dependence between Q and Q_1.

Case 2: This shows that Q_1 modifies a database-part and Q accesses only a part of it. Therefore, Q is dependent on Q_1.

Figure 6: Refined DOPDG (of Figure 5) applying Condition-Action rules [4] (data dependencies $6 \xrightarrow{salary} 7$ and $6 \xrightarrow{salary} 8$ are removed).

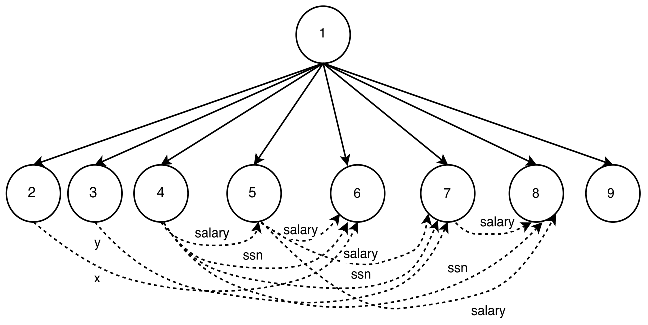
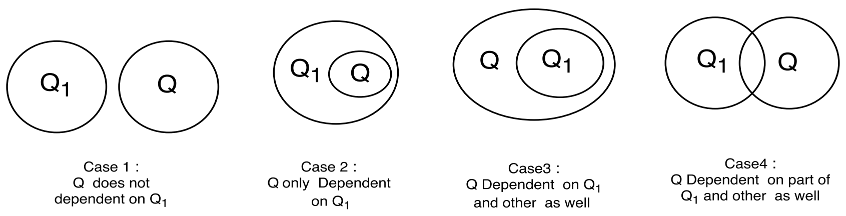


Figure 7: Overlapping of data defined by Q_1 and used by Q



Q1
Q
Case 1 : Q does not dependent on Q1
Case 2 : Q only Dependent on Q1
Case3 : Q Dependent on Q1 and other as well
Case4 : Q Dependent on part of Q1 and other as well

Case 3: This shows that Q_1 modifies a part of the database and Q accesses the modified-part as well as some unmodified part of the database db. Therefore, Q is dependent on both Q_1 and db.

Case 4: This shows that Q_1 modifies a database-part and Q accesses a part of it as well as some unmodified part of the database db. Therefore, Q is dependent on both Q_1 and db.

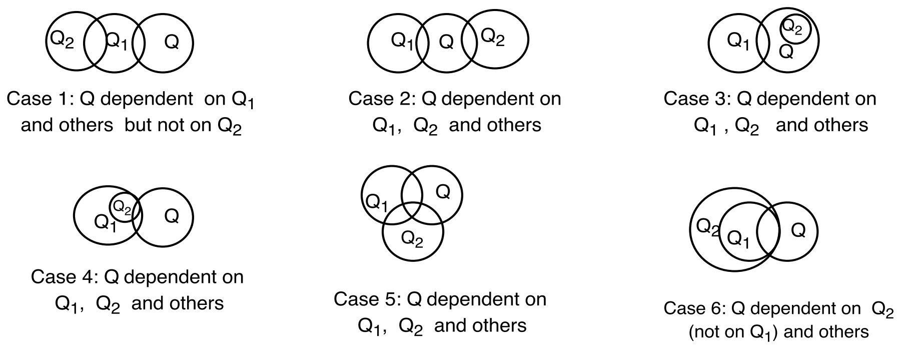
Let us extend the same for two definitions: suppose Q_2 and Q_1 be two database statements defining an attribute x which is subsequently used by Q. The overlapping of defined- and used-part of x-data is represented by Venn diagram shown in the Figure 8, explained below:

Case 1: This shows that Q_1 modifies a database-part and Q_2 partially re-modifies database-part which is modified by Q_1 and also modifies other database-part. Q accesses only part of database-part which is modified by Q_1 and others database-part, but not from database-part which is modified by Q_2. Therefore, Q dependent on Q_1 and db but not Q_2.

Case 2: This shows that Q_1 and Q_2 modifies different database-part and Q accesses partially database-part modified by Q_1 and Q_2 and others. Therefore, Q dependent on Q_1, Q_2 and db.

Case 3: This shows that Q_1 and Q_2 modifies different database-part and Q accesses Q_2 fully, Q_1 and other database-part partially. Therefore, Q dependent on Q_1, Q_2 and db.

Figure 8: Overlapping of data defined by Q_2, Q_1 (in sequence) and used by Q


Q2 Q1 Q
Case 1: Q dependent on Q1 and others but not on Q2
Q1 Q Q2
Case 2: Q dependent on Q1, Q2 and others
Q1 Q2 Q
Case 3: Q dependent on Q1 , Q2 and others
Q2 Q1 Q
Case 4: Q dependent on Q1, Q2 and others
Q1 Q Q2
Case 5: Q dependent on Q1, Q2 and others
Q2 Q1 Q
Case 6: Q dependent on Q2 (not on Q1) and others

Case 4: This shows that Q_1 modifies database-part which is partially re-modified by Q_2 and Q accesses partially database-part modified by Q_1 and Q_2 and db. Therefore, Q dependent on Q_1, Q_2 and db.

Case 5: This shows that Q_1 modifies database-part and Q_2 partially modifies database-part modified by Q_1 and other database part. Q accesses partially database-part modified by Q_2, Q_1 and db. Therefore, Q dependent on Q_1, Q_2 and db.

Case 6: This shows that Q_1 modifies a part of database which is then fully re-modified by Q_2 and finally accesses by Q. Therefore, Q dependent on Q_2 (but not on Q_1) and db.

Disadvantage of Condition-Action Rules: The Condition-Action rules [4] can be applied only on a single def-use pair at a time. In case of more than one defining database statements where all partially define (in sequence) attribute x which is then used by another statement, the Condition-Action rules fail to capture semantic independencies. For instance, Condition-Action rules can not identify the semantic independencies of Q on Q_1 depicted in case 6 of figure 8. Example 6 depicts this using an example code.

Example 6: Consider the database application depicted in Figure 9. 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 10(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 10(b).

4 Data-centric Refinement of DOPDG

In this section, we propose a refinement of dependence graph based on the value of attributes (instead of syntactic presence of attributes). Our proposed algorithm consist of five phases listed below:

- Phase 1: Construction of Action Tree.
- Phase 2: Computing traces.

Figure 9: Program code P and database table t

(a) Program code P

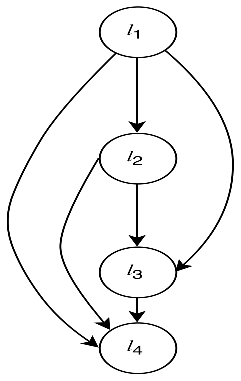
```
ℓ1.   Statement myStmt = DriverManager.getConnection("jdbc mysql:200. 210. 220.
      1:1114/demo","scott","tiger").createStatement();
ℓ2.   UPDATE a = a + 5 WHERE a BETWEEN 20 AND 50;
ℓ3.   UPDATE a = a + 1 WHERE a BETWEEN 10 AND 55;
ℓ4.   SELECT a WHERE a BETWEEN 10 AND 60;
```

(b) Table t

id	a	b
1	10	5
2	50	9
3	30	15
4	20	40
5	60	30
6	70	10
7	40	20
8	80	60
9	90	80
10	100	75

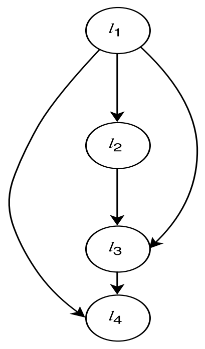
Figure 10: Refinement failure by Condition-Action rules

(a) Dependencies by applying Condition-Action rules [4] on the DOPDG of code in Figure 9a



$$\begin{tikzcd}[row sep=large] l_1 \arrow[d] \arrow[dd, bend left] \arrow[ddd, bend right] \\ l_2 \arrow[d] \arrow[dd, bend right] \\ l_3 \arrow[d] \\ l_4 \end{tikzcd}$$

(b) Expected more precise DOPDG



$$\begin{tikzcd}[row sep=large] l_1 \arrow[d] \arrow[dd, bend left] \arrow[ddd, bend right] \\ l_2 \arrow[d] \\ l_3 \arrow[d] \\ l_4 \end{tikzcd}$$

- Phase 3: Backwards with precondition.
- Phase 4: Product of traces and observational-window.
- Phase 5: Identifying dependencies.

Let us describe in details each of the phases.

4.1 Phase 1: Construction of Action Tree.

This phase is based on the partitioning of database states using condition-part present in the defining database statements (*e.g.*, INSERT, DELETE and UPDATE). Database state is the current state of the database after creating and populating it for a particular purpose. After each operation performed on the database by the application program, database state changes either partially or completely. The part of database to be modified depends on the condition presents in the database definition statement.

Given a database statement Q, the abstract syntax is denoted by $Q = \langle A, \phi \rangle$ where A and ϕ denotes the action-part and condition-part of Q respectively [4]. For instance, the query "SELECT a_1, a_2 FROM t WHERE $a_3 \leq 30$" is denoted by $\langle A, \phi \rangle$ where A represents the action-part "SELECT a_1, a_2 FROM t" and ϕ represents the conditional-part "$a_3 \leq 30$". Database part defined by a database modification statement(*e.g.*, INSERT, DELETE and UPDATE) represented by a node. 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 root node of the Action Tree is a special node that represents initial state of database. Order of the database definition statements in the application program represents level of nodes in the Action Tree. The following example explains the construction of Action Tree in detail.

Example 7: Let us consider the program code P shown in Figure 9(a). Analysing the application program, the sequence of definitions of attribute 'a' can be represented in a fixed order: $\ell_1 \rightarrow \ell_2 \rightarrow \ell_3$.

The root of the Action Tree represents the initial database state defined at program point ℓ_1 (*i.e.* this statement acts as a defining statement for the values of all attributes in DB). We denote this definition as action part 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 labelled by "ℓ_1 : true".

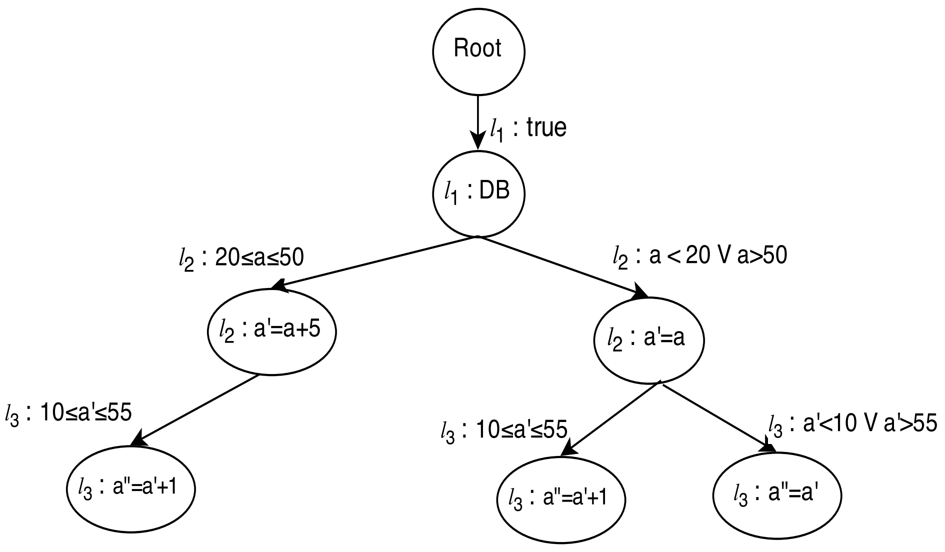
Consider the next defining statement at ℓ_2. The condition-part "$\phi = 20 \leq a \leq 50$" divides the database into two parts: one satisfies ϕ (say, P_ϕ) and other satisfies $\neg\phi$ (say, $P_{\neg\phi}$). The action "a = a + 5" is applied on P_ϕ, whereas $P_{\neg\phi}$ remains same. As according to the Condition-Action rules, both $\ell_1 \xrightarrow{P_\phi} \ell_2$ and $\ell_1 \xrightarrow{P_{\neg\phi}} \ell_2$ exist, we crate two children nodes – one denotes the action "ℓ_2 : a' = a + 5" (the edge labelled by $\ell_2 :\ \phi$ *i.e.* $\ell_2 :\ 20 \leq a \leq 50$) and other denotes the action "ℓ_2 : a' = a (the edge labelled by $\ell_2 :\ \neg\phi$ *i.e.* $\ell_2 :\ a < 20 \wedge a > 50$). Note that we rename the attributes after an action is performed to distinguish the values before and after the action.

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 11.

4.2 Phase 2: Computing traces.

A trace in an Action Tree is a path from the root node to a leaf node. For a given Action Tree shown in the Figure 11, we compute all traces from its root node to all leaves nodes.

Figure 11: Action Tree of program code P shown in Figure 9a.



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 8: Given the Action Tree in Figure 11, we compute all traces traversing the tree from the root node to all leaves nodes. These are shown in Table 2. Here $\langle \tau_i \rangle_{i \geq 1}$ is the Action Tree traces , $\langle\, (\ell_i : \phi,\ \ell_i : A)$ is an element of a Action Tree trace τ_i.

Table 2 Traces of Action Tree shown in the figure 11

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

4.3 Phase 3: Backwards with precondition.

Hoare logic [8] is a deductive system whose axioms and rules of inference provides 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 the 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, $\{j = 3 \wedge k = 4\} j := j + k \{j = 7 \wedge k = 4\}$. Here $j = 3 \wedge k = 4$ is the "precondition", $j := j + k$ is the statement and $j = 7 \wedge k = 4$ is the "postcondition" of Hoare triple.

As our objective is to find all semantics-based dependencies 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, in the running example (Figure 9), the observation window is the condition part of the used-statement at ℓ_4, *i.e.* $10 \leq a'' \leq 60$.

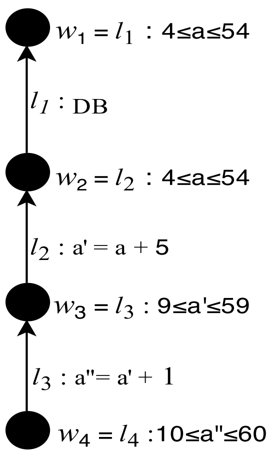
Considering the observational-window as postcondition, we apply Hoare logic to compute weakest precondition at each program point appears before. Our aim is to determine the flow of definitions and to verify whether it affects the observational-window. Let us illustrate using running example.

Example 9: Consider the program P shown in Figure 9. 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 in Figure 12.

Figure 12: Weakest precondition at each program point of P in Figure 9a applying Hoare logic



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).

4.4 Phase 4: 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:

$$\tau \times \tau_o = \langle\ (\ell_i : \phi',\ \ell_i : A')\ \rangle_{i\geq 1} \times \langle(\ell_j : \phi'')\rangle_{j\geq 1} = \langle\ (\ell_x : u,\ \ell_x : v)\rangle_{x\geq 1}, \text{ where}$$

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

Example 10: Consider the running example. The product of the Action Tree traces (τ_1, τ_2, τ_3) and observational-window trace (τ_o) results the following:

Table 3 Product of Action Tree traces and observational-window trace.

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

4.5 Phase 5: Identifying dependencies.

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 dependencies:

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

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

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

Example 11: In our running example, consider the set of traces obtained after performing product operation in Table 3. The set of masked traces is shown in Table 4(a). The result of applying Filter on the masked traces is shown in Table 4(b). Applying the function DEP on the filtered masked traces, we get the semantics-based dependencies depicted in Table 4(c).

Table 4 Identifying semantics-based dependencies

(a) Masked traces

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

(b) Applying Filter on masked traces

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

(c) Semantics-based dependencies

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

5 Enhancement using Binary Decision Diagram

The first phase of our proposed algorithm in section 4 results an Action Tree whose time and space complexities are exponential with respect to the number of defining statements for an attribute. At each level of the tree, the algorithm adds the condition and its negation present in the WHERE clause of the next defining statement. In this section, we propose a reduction of the complexities using Binary Decision Diagram (BDD) [2].

Binary Decision Diagram (BDD) [2]: A Binary Decision Diagram (BDD) for a boolean function is a finite directed acyclic graph with unique initial node, where

- All terminal nodes are labelled with either 0 or 1,
- All non-terminal nodes are labelled with a boolean variable, and
- Each non-terminal node has exactly two outgoing edges: one labelled with 0 represented by dashed line and the other labelled with 1 represented by solid line.

Binary Decision Diagram can further be compacted by applying the following reduction rules.

- Eliminating duplicate terminal: If a BDD contains more than one terminal 0-node, redirect all edges which point to such a 0-node to just one of them. Similar is performed for all nodes labelled with 1.
- Eliminating redundant node: If outgoing edges of a node points to the same node for different value of boolean variable then it is called redundant node.
- Merge duplicate non-terminal node: The out going edges of two or more unique nodes points to the same node are called duplicate nodes.

Two or more terminal nodes of binary decision diagram having same label can be grouped into a single node by using reduction rules of BDD. The resultant BDD after reducing nodes following repeatedly all the three reduction rules is called Reduced Binary Decision

Diagram (RBDD). RBDD may not be unique for a given set of boolean variables as we do not consider any order among the variables. Ordering among the boolean variable without repeatation results an unique RBDD that is called Ordered Reduced Binary Decision Diagram (ORBDD).

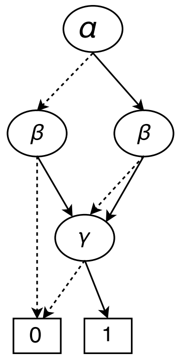
Due to compactness of BDD, it has a large application areas such as correctness of combinational circuit, model checking, program analysis, automatic test generation, artificial intelligence, data mining, software security, fault tolerance computing , abstract interpretation, and so on [2]. The following example illustrates the reduction of BDD.

Example 12: Given the boolean expression $\Gamma = (\alpha\gamma + \beta\gamma)$ and its truth table shown below:

α	β	γ	Γ
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

The Reduced Binary Decision Diagram of Γ based on the above truth table is shown in Figure 13.

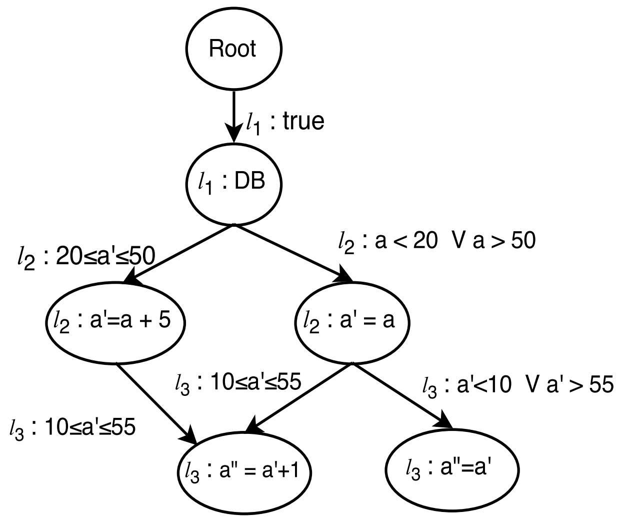
Figure 13: Reduced Binary Decision Diagram of Γ



We are now in a position to extend the notion of RBDD to our algorithm to reduce the space and time complexity. The following example illustrates this.

Example 13: Consider the Action Tree shown in the Figure 11. Observe that the Action Tree contains many duplicate nodes with same labels. Therefore they can be grouped into one by applying reduction rules of BDD. For instance, the Action Tree in figure 11 can be reduced to Reduced Action Tree depicted in Figure 14. Interestingly, the ordering among defining database statements according to the program points yields a unique Reduced Action Tree which we call Ordered Reduced Action Tree.

Figure 14: Ordered Reduced Action Tree



For a large Action Tree, these reduction rules reduces the number of terminal nodes and non-terminal nodes from the Action Tree - hence the space complexity. As the ordered reduced Action Tree has unique $n + 1$ nodes, the construction time is also reduced to $O(n)$, where n is the number of nodes in the Action Tree.

6 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 $x \overset{*}{\rightarrow} y$ 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 database 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 variable presents in the slice with *high* security level, the analysis reports a possible information leakage.

Example 14: Consider a company which maintains a database of its customers and label "Status"(*i.e.* Normal, Silver and Gold) to each customer based on the total transaction in a month. Suppose that the company gives offers to its customers who have purchased items costs more than rupees 10,000 in a month. It also gives additional special offer of rupees 200 to the customers whose status is "Silver" and of Rupees 500 to the customer whose status is "Gold". Let us consider the database program AP and the underlying table Customer shown in Figures 15(a) and 15(b) respectively.

Suppose, the attributes ID, CustName, Address and OfferAmt have low security level, whereas TransAmt has high security level. The refined DOPDG of program AP based on our proposed method is shown Figure 15(c). As there are two consecutive definitions of attribute "OfferAmount" at program points 8 and 9, the dependence $8 \xrightarrow{rs.OfferAmt} 11$ exists even after applying Action-Condition rules. On the other hand, our method removes the false dependence $8 \xrightarrow{rs.OfferAmt} 11$, because the value of "OfferAmt" defined by 8 is redefined completely at program point 9.

The slicing of the DOPDG *w.r.t.* slicing criterion c = ⟨11, {ID, OfferAmt}⟩ produces a slice: $\{2, 4, 5, 6, 8, 9\}$. As the slicing includes the statement 5 involving *high* variable "TransAmt", this means that there exists a path $5 \overset{*}{\rightarrow} 11$ which indicates a flow of secure information about "TransAmt" to the output channel. Therefore, the program is not secure.

7 Complexity and Correctness

In this section, we discuss the complexity of our proposal and prove the correctness of the algorithm.

7.1 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. Therefore, the time complexity to compute all traces from the root node to all leaves nodes is $O(2^n)$. 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 ≪n. Further, time and space complexity can be reduced to linear by applying reduction rules of BDD on Action Tree.

Figure 15: Information Flow Analysis on Refined DOPDG of AP

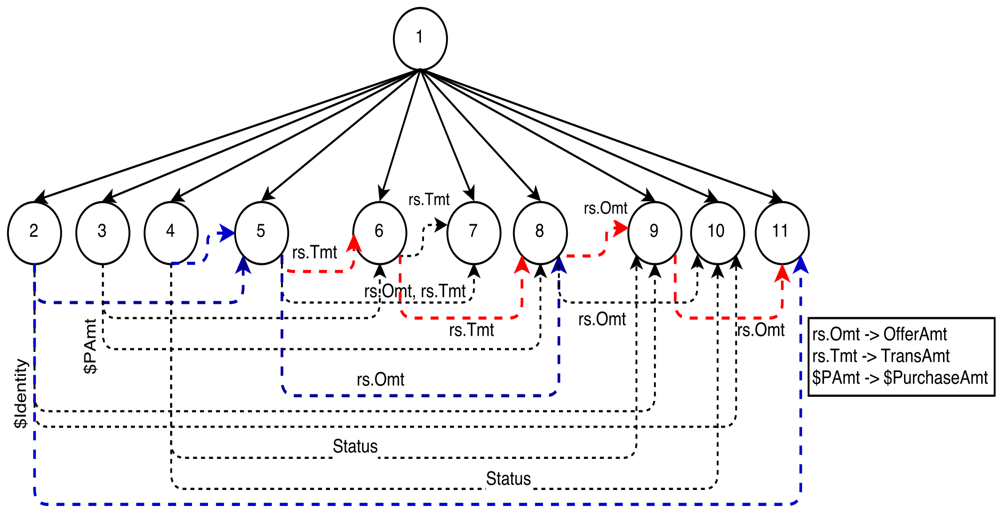
(a) Application Program AP

```
Procedure Compute_Offer_Amount
1.   Start;
2.   $Identity = input();
3.   $PurchaseAmt = getPurchase( );
4.   Statement myStmt = DriverManager.getConnection(“jdbc mysql:…… /demo” “X",“Y").createStatement();
5.   ResultSet rs = myStmt.excecuteQuery(“SELECT TransAmt, OfferAmt FROM Customer WHERE ID = $identity”);
6.   UPDATE Customer SET rs.TransAmt = rs.Transamt + $PurchaseAmt;
7.   UPDATE Customer SET rs.OfferAmt = 0.0 WHERE rs.TransAmt ≤ 10000;
8.   UPDATE Customer SET rs.OfferAmt = 0.10 * $PurchaseAmt WHERE rs.TransAmt > 10000;
9.   UPDATE Customer SET rs.OfferAmt = rs.OfferAmt + 200 WHERE Status = ‘Silver’ AND WHERE ID = $identity;
10.  UPDATE Customer SET rs.OfferAmt = rs.OfferAmt + 500 WHERE Status = ‘Gold’ AND WHERE ID = $identity;
11.  SELECT ID, OfferAmt WHERE ID = $Identity AND Status = ‘Silver’;
```

(b) Table Customer

ID	CustName	Address	TransAmt	OfferAmt	Status
101	David	Denve, Colorado	20000	250	Silver
102	William John	Albani, New york	15000	170	Silver
103	Rajesh Kumar	Patna, Bihar	24000	310	Gold
104	Prity Agrawal	Baglore, Karnataka	10000	155	Silver
105	Jacob Smith	Austin, Texas	12500	165	Silver
106	Niraj Sinha	Mumbai, Maharastra	2500	00	NULL

[illegible]



7.2 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 $t = e_1e_2e_3......e_{n-1}e_n$ such that DEP(t) = DEP($e_1e_2e_3......e_{n-1}e_n$) = Label($e_{n-1}$) $\longrightarrow$ Label(e_n) = l $\rightarrow$ k where e_{n-1} = (l : true, l : yes), e_n = (k: ϕ, k: observe) and ϕ is the observational-window. If we move backward along the phases of the proposed "true" and " yes" in e_{n-1} indicates the following: $\exists$ t_a ϵ action-tree trace and $\exists$ t_o ϵ 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} elements from the trace, and $\phi_a \bigwedge \phi_o \neq \emptyset$. 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 $\phi_a \bigwedge \phi_o \neq \emptyset$ therefore $D \cap U \neq \emptyset$.

8 Conclusions and Future Works

In this paper, we proposed a dependency refinement algorithm for database applications that computes dependencies among database statements semantically rather than syntactically. This leads to an improvement of the precision of DOPDG-based information flow analysis. The proposal covers a more generic scenario where attackers are allowed to observe a part of insensitive database information (rather than all) corresponding to public attributes. Although the time complexity of the proposed algorithm is exponential as database attributes map to a list of values (rather than single value), we have shown how the use of Ordered Reduced Binary Decision Diagram can effectively reduce the time complexity. As a future work, we are implementing a prototype of the proposed work to experiment the precision and efficiency of our proposed approach on real benchmark codes.

Acknowledgement

This work is partially supported by the research grant (SB/FTP/ETA-315/2013) from the Science & Engineering Research Board (SERB), Department of Science and Technology, Government of India. We thank the anonymous reviewers and Editor-in-Chief of this journal for their valuable comments and helpful suggestions.

References

[1] Torben Amtoft and Anindya Banerjee. A logic for information flow analysis with an application to forward slicing of simple imperative programs. Science of Computer Programming, 64(1):3–28, 2007.

[2] Henrik Reif Andersen. An introduction to binary decision diagrams.

[3] Gregory R. Andrews and Richard P. Reitman. An axiomatic approach to information flow in programs. ACM Trans. Program. Lang. Syst., 2:56–76, 1980.

[4] Elena Baralis and Jennifer Widom. An algebraic approach to rule analysis in expert database systems. 1994.

[5] Salvador Cavadini. Secure slices of insecure programs. In Proc. of the ACM symposium on Information, computer and communications security, pages 112–122, Tokyo, Japan, 2008. ACM Press.

[6] Agostino Cortesi and Raju Halder. Information-flow analysis of hibernate query language. In Proc. of the 1st Int. Conf. on Future Data and Security Engineering, pages 262–274, Vietnam, 2014. Springer LNCS 8860.

[7] Dorothy E Denning. A lattice model of secure information flow. Communications of the ACM, 19(5):236–243, 1976.

[8] Edsger Wybe Dijkstra and Dijkstra. A discipline of programming, volume 1. prentice-hall Englewood Cliffs, 1976.

[9] Rayna Dimitrova, Bernd Finkbeiner, Máté Kovács, Markus N Rabe, and Helmut Seidl. Model checking information flow in reactive systems. In Verification, Model Checking, and Abstract Interpretation, pages 169–185. Springer, 2012.

[10] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems (TOPLAS), 9(3):319–349, 1987.

[11] Joseph A Goguen and José Meseguer. Unwinding and inference control. In Security and Privacy, 1984 IEEE Symposium on, pages 75–75. IEEE, 1984.

[12] Raju Halder and Agostino Cortesi. Abstract interpretation of database query languages. Computer Languages, Systems & Structures, 38:123–157, 2012.

[13] Raju Halder, Matteo Zanioli, and Agostino Cortesi. Information leakage analysis of database query languages. In Proceedings of the 29th Annual ACM Symposium on Applied Computing, pages 813–820. ACM Press, 2014.

[14] Christian Hammer, Jens Krinke, and Gregor Snelting. Information flow control for java based on path conditions in dependence graphs. In IEEE International Symposium on Secure Software Engineering, pages 87–96, 2006.

[15] Christian Hammer and Gregor Snelting. 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.

[16] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems (TOPLAS), 12(1):26–60, 1990.

[17] Sebastian Hunt and David Sands. On flow-sensitive security types. In Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006), pages 79–90. ACM Press, Charleston, South Carolina, USA, January 11–13 2006.

[18] Rajeev Joshi and K Rustan M Leino. A semantic approach to secure information flow. Science of Computer Programming, 37(1):113–138, 2000.

[19] Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In Proceedings of the 18th International Conference on Software Engineering, pages 495–505, Berlin, Germany, 1996. IEEE CS.

[20] Alexander Lux and Heiko Mantel. Who can declassify? In Proceedings of the 5th International Workshop on Formal Aspects in Security and Trust (FAST 2008), pages 35–49. Springer LNCS 5491, Malaga, Spain, October 9–10 2008.

[21] Heiko Mantel and Henning Sudbrock. Types vs. pdgs in information flow analysis. In Proceedings of the 23rd International Symposium on Logic-Based Program Synthesis and Transformation, pages 106–121, Madrid, Spain, 2013. Springer LNCS 7844.

[22] Einar Mykletun, Maithili Narasimha, and Gene Tsudik. Authentication and integrity in outsourced databases. ACM Transactions on Storage (TOS), 2(2):107–138, 2006.

[23] Mohapatra Durga Prasad, Rajib Mall, and Rajeev Kumar. An overview of slicing techniques for object-oriented programs. Informatica (Slovenia), 30(2):253–277, 2006.

[24] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. Selected Areas in Communications, IEEE Journal on, 21(1):5–19, 2003.

[25] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. Journal of Computer Security, 17:517–548, 2009.

[26] Geoffrey Smith. Principles of secure information flow analysis. In Malware Detection, pages 291–307. Springer, 2007.

[27] Mana Taghdiri, Gregor Snelting, and Carsten Sinz. Information flow analysis via path condition refinement. In Formal Aspects of Security and Trust, pages 65–79. Springer, 2011.

[28] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. Journal of computer security, 4(2):167–187, 1996.

[29] Daniel Wasserrab, Denis Lohner, and Gregor Snelting. On pdg-based noninterference and its modular proof. In Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security, pages 31–44. ACM, 2009.

[30] Wei Wei and Ting Yu. Integrity assurance for outsourced databases without dbms modification. In Proceedings of the 28th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy XXVIII (DBSec 2014), pages 1–16. Springer LNCS 8566, Vienna, Austria, July 14–16 2014.

[31] David Willmor, Suzanne M Embury, and Jianhua Shao. Program slicing in the presence of database state. In Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on, pages 448–452. IEEE, 2004.

[32] Matteo Zanioli and Agostino Cortesi. Information leakage analysis by abstract interpretation. In SOFSEM 2011: Theory and Practice of Computer Science, pages 545–557. Springer, 2011.

[33] Matteo Zanioli, Pietro Ferrara, and Agostino Cortesi. Sails: static analysis of information leakage with sample. In Proceedings of the 27th Annual ACM Symposium on Applied Computing, pages 1308–1313. ACM, 2012.