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

Copyright © 2016 Inderscience Enterprises Ltd.
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 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:

$$\sigma_1 \equiv_{\text{low}} \sigma_2 \implies \llbracket P \rrbracket_{\sigma_1} \equiv_{\text{low}} \llbracket P \rrbracket_{\sigma_2}$$

(1)

where $P$ is a program or program statement, $\sigma_1$ and $\sigma_2$ ($\sigma_1 \neq \sigma_2$) are two states of $P$ before running it, $\llbracket P \rrbracket_{\sigma_1}$ and $\llbracket P \rrbracket_{\sigma_2}$ are the states obtained after running the program $P$ on $\sigma_1$ and $\sigma_2$ respectively. $\sigma_1 \equiv_{\text{low}} \sigma_2$ 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
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 ≥ 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 high 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:

```plaintext
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 ⊢ Γ{c}Γ’ is derivable.

Various forms of dependence graphs (Program Dependence Graph (PDG), System Dependence Graph (SDG), Class Dependence Graph (CIDG), 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,
Refining Dependencies for Inf. Flow Analysis of DB Applications

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. \( x = \text{getHighInput}(); \)
3. \( y = x \times 5; \)
4. \( z = 5 \times y \div 5; \)
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 \( \mathbf{e} = \langle e_1, e_2, \ldots, e_n \rangle \) represents a sequence of arithmetic expressions, \( \mathbf{v}_d = \langle v_{d_1}, v_{d_2}, \ldots, v_{d_m} \rangle \) 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 \( \phi \) 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 | ∈ | K (Constants) |
| t_j | ∈ | V_j (Database Attributes) |
| v | ∈ | V = V_r ∪ V_s (All Variables) |
| v | ∈ | W (Well-formed Formulas in First Order Logic) |
| c | ∈ | k | t_j | t_k | c_1 @ c_2; c_1 ∈ {>, ≥, ≤, …} |
| b | ∈ | e_1 = e_2 | c_1 @ c_2; c_1 ∈ {>, ≥, ≤, …} |
| g(\beta) | ∈ | GROUP BY(\beta) | id |
| r | ∈ | DISTINCT | ALL |
| s | ∈ | AVG | SUM | MAX | MIN | COUNT |
| b(c) | ∈ | s ÷ r | DISTINCT(q) | id |
| h(c) | ∈ | COUNT(|(h_1(x_1),…,h_n(x_n)), where h = (h_1,...,h_n) and x = (x_1,...,x_n) |
| f(c) | ∈ | ORDER BY ASC(\beta) | ORDER BY DESC(\beta) | id |
| Q ⊃ Q | ∈ | (A, φ) |
| \phi(V) | ∈ | SELECT(c_1, f(\beta), h(\beta)), φ, g(\beta) | UPDATE(c_1, \beta) | INSERT(c_1, \beta) | DELETE(c_1) |

\langle n_1, …, n_k \rangle ∈ N^* where (n_i, n_{i+1}) ∈ E for all i ∈ {1, …, k − 1}. A path is called trivial if it is of the form (n) (i.e., a sequence of length 1), and non-trivial otherwise. A node n is on the path \langle n_1, …, n_k \rangle if n = n_i for some i ∈ {1, …, 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: \( N → \phi(V) \) are functions returning the defined and used variables set, respectively, for a node.

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

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

Control Flow Graph (CFG): The Control Flow Graph of c ∈ Com is CFG_c = (N_c, E_c, def_c, use_c), where the set of nodes is defined as N_c = {1, …, |c|} ∪ {start, stop} and the set of edges E_c ⊆ N_c × N_c is defined recursively by:

- \( E_{\text{skip}} = E_{v_a := e} = E_{v_a := ?} = E_Q = E_{\text{output } e} = \{(\text{start}, 1), (1, \text{stop})\} \)
- \( E_{if(b) \ \text{then } c_1 \ \text{else } c_2 \ |f| = \{(\text{start}, 1), (\text{start}, \text{stop})\} \)
- \( E_{while(b) \text{ do } c \ \text{od} |f| = \{(\text{start}, 1), (\text{start}, \text{stop})\} \)
- \( E_{c_1; c_2 |f| = \{(\text{start}, 1), (\text{start}, \text{stop})\} \)
- \( E_{v_a := e |f| = \{(\text{start}, 1), (\text{start}, \text{stop})\} \)
- \( E_{v_a := ? |f| = \{(\text{start}, 1), (\text{start}, \text{stop})\} \)
- \( E_{Q |f| = \{(\text{start}, 1), (\text{start}, \text{stop})\} \)
- \( E_{\text{output } e |f| = \{(\text{start}, 1), (\text{start}, \text{stop})\} \)
- \( E_{\text{def}_c = E_{\text{use}_c} = E_{\text{start} := e} = E_{\text{stop} := ?} = E_Q = E_{\text{output } e} = \{(\text{start}, 1), (1, \text{stop})\} \)
- \( E_{c_1; c_2 |f| = \{(\text{start}, 1), (\text{start}, \text{stop})\} \)
- \( E_{while(b) \text{ do } c \ \text{od} |f| = \{(\text{start}, 1), (\text{start}, \text{stop})\} \)
- \( E_{c_1; c_2 |f| = \{(\text{start}, 1), (\text{start}, \text{stop})\} \)
- \( E_{\text{def}_c = E_{\text{use}_c} = E_{\text{start} := e} = E_{\text{stop} := ?} = E_Q = E_{\text{output } e} = \{(\text{start}, 1), (1, \text{stop})\} \)}
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 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:

\[
\begin{align*}
def_c(2) &= \{x\} \\
def_c(3) &= \{y\} \\
def_c(4) &= \{\text{ssn}, \text{name}, \text{salary}\} \\
def_c(5) &= \{\text{salary}, i\} \\
use_c(5) &= \{\text{salary}\} \\
def_c(6) &= \{\text{salary}, i\} \\
use_c(6) &= \{\text{salary}, \text{ssn}, x\}
\end{align*}
\]

<table>
<thead>
<tr>
<th>ssn</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John</td>
<td>20000</td>
</tr>
<tr>
<td>2</td>
<td>Smith</td>
<td>14500</td>
</tr>
<tr>
<td>3</td>
<td>David</td>
<td>24000</td>
</tr>
<tr>
<td>4</td>
<td>Marry</td>
<td>30000</td>
</tr>
<tr>
<td>5</td>
<td>Kumar</td>
<td>25000</td>
</tr>
<tr>
<td>6</td>
<td>Arun</td>
<td>20000</td>
</tr>
<tr>
<td>7</td>
<td>Shayam</td>
<td>16000</td>
</tr>
<tr>
<td>8</td>
<td>Lisa</td>
<td>15000</td>
</tr>
<tr>
<td>9</td>
<td>Manoj</td>
<td>30000</td>
</tr>
<tr>
<td>10</td>
<td>Kumar</td>
<td>24000</td>
</tr>
</tbody>
</table>
Given \( c \) defined-variables to construct more precise DOPDG. The algorithm simply identifies

\[ \text{def}, (7) = \{ \text{salary, } i \} \]
\[ \text{use}, (7) = \{ \text{salary, } ssn, y \} \]
\[ \text{use}, (8) = \{ \text{salary, } ssn \} \]

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 \( c \in \text{Com} \) such that \( N_c = \{ 1, \ldots, |c| \} \cup \{ \text{start, stop} \} \), let us define an improved version of the two functions \( \text{def}_c, \text{use}_c \):

\[
\text{def}_c : N_c \rightarrow \varphi(\mathbb{V} \times \text{Lab})
\]
\[
\text{use}_c : N_c \rightarrow \varphi(\mathbb{V} \times \text{Lab})
\]

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 \( \text{Lab} = [\text{full}, \text{partial}] \). For \( n \in \{ 1, \ldots, |c| \} \),

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

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

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
Refining Dependencies for Inf. Flow Analysis of DB Applications

defined when there is no WHERE clause in INSERT, DELETE, UPDATE statements (i.e. \( \phi \) acts as tautology). Observe that, in case of application variable, the label is by default “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:

\[
\begin{align*}
\text{def}_c(1) &= \{(x, \text{partial})\} \\
\text{def}_c(2) &= \{(x, \text{full})\} \\
\text{use}_c(3) &= \{(x, \text{partial})\}
\end{align*}
\]

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

\[
\begin{align*}
\text{def}_c(1) &= \{(x, \text{partial})\} \\
\text{def}_c(2) &= \{(x, \text{partial})\} \\
\text{use}_c(3) &= \{(x, \text{partial})\}
\end{align*}
\]

\( Q_3 \) 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:

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

According to this information, we observe that all values of attribute ‘salary’ are updated at program point 5, resulting the data-dependencies \( 4 \rightarrow 6, 4 \rightarrow 7, 4 \rightarrow 8 \) 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 $\text{def}_c$ and $\text{use}_c$ in equations 2 and 3 (data-dependencies $4 \rightarrow 6$, $4 \rightarrow 7$, $4 \rightarrow 8$ 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:

$$E_{\text{cond}} \rightarrow E_{\text{act}}$$

Where $E_{\text{cond}}$ and $E_{\text{act}}$ 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 $E^X_{\text{cond}} \rightarrow E^X_{\text{upd}}$:

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

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

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
Refining Dependencies for Inf. Flow Analysis of DB Applications

and an interest rate less than 3% is increased by 1%. The rule is expressed as $E^Y_{cond} \rightarrow E^Y_{upd}$ with

$$E^Y_{cond} = \Pi_{city,C}(\sigma_{C > 5000}(\sigma_{C = count(name)})(A_{city=SF\text{-CUST}}))$$

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

$$E^Y_c = \sigma_{bal > 5000 \land rate < 3}ACCT \sqcap_{\text{name}} \sigma_{city=SF\text{-CUST}}$$

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

$$C = E^X_{cond} = \sigma_{bal < 500 \land rate > 0}ACCT$$

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

$$E^Y_c = \sigma_{bal > 5000 \land rate < 3}ACCT \sqcap_{\text{name}} \sigma_{city=SF\text{-CUST}}$$

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 and use, 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 $6 \rightarrow 7$ and $6 \rightarrow 8$ 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 $6 \rightarrow 7$ and $6 \rightarrow 8$ are removed).

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

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 $db$. 
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. $\ell_1 \rightarrow \ell_2$, $\ell_1 \rightarrow \ell_3$, $\ell_1 \rightarrow \ell_4$, $\ell_2 \rightarrow \ell_3$, etc.), is depicted in Figure 10(a). However, observe that, since $\ell_4$ uses the values within the range 10 to 60 of ‘$a$’, no values defined by statement $\ell_2$ can directly affect the observing range of $\ell_4$, because $\ell_3$ redefines the values. Therefore, the dependence $\ell_2 \rightarrow \ell_4$ 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

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

<table>
<thead>
<tr>
<th>id</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>60</td>
<td>30</td>
</tr>
<tr>
<td>6</td>
<td>70</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>80</td>
<td>60</td>
</tr>
<tr>
<td>9</td>
<td>90</td>
<td>80</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>75</td>
</tr>
</tbody>
</table>

Figure 10: Refinement failure by Condition-Action rules

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

(b) Expected more precise DOPDG

• 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 \(\phi\) 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 \(\phi\) 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 \(\ell_1\) (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 “\(\ell_1 : DB\)” connected with the root node by an edge. As there is no condition-part involved, the edge is labelled by “\(\ell_1 : true\)”.

Consider the next defining statement at \(\ell_2\). The condition-part “\(\phi = 20 \leq a \leq 50\)” divides the database into two parts: one satisfies \(\phi\) (say, \(P\phi\)) and other satisfies \(\neg\phi\) (say, \(P\neg\phi\)). The action “\(a = a + 5\)” is applied on \(P\phi\), 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 labelled by \(\ell_2 : \phi\) i.e. \(\ell_2 : 20 \leq a \leq 50\)) and other denotes the action “\(\ell_2 : a' = a\)” (the edge labelled by \(\ell_2 : \neg\phi\) i.e. \(\ell_2 : a < 20 \land a > 55\)). 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 \(\ell_3\) on the result just obtained. Observe that as \(\ell_2 \xrightarrow{P\neg\phi} \ell_3\) 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 $\phi \in W$ (set of well-formed formulas) and $A \in \mathcal{A}$ (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, $\langle (\ell_i : \phi, \ell_i : A) \rangle$ is an element of a Action Tree trace $\tau_i$.

Table 2  Traces of Action Tree shown in the figure 11

\[
\begin{align*}
\tau_1 &= \langle \ell_1 : \text{true}, \ell_1 : \text{DB} \rangle \langle \ell_2 : 20 \leq a \leq 50, \ell_2 : a = a + 5, \ell_3 : 10 \leq a' \leq 55, \ell_3 : a' = a' + 1 \rangle \\
\tau_2 &= \langle \ell_1 : \text{true}, \ell_1 : \text{DB} \rangle \langle \ell_2 : a < 20 \lor a > 50, \ell_2 : a' = a, \ell_3 : 10 \leq a' \leq 55, \ell_3 : a' = a' + 1 \rangle \\
\tau_3 &= \langle \ell_1 : \text{true}, \ell_1 : \text{DB} \rangle \langle \ell_2 : a < 20 \lor a > 50, \ell_2 : a' = a, \ell_3 : a' < 10 \lor a' > 55, \ell_3 : a' = a' \rangle
\end{align*}
\]

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, $\{j = 3 \land k = 4\} j := j + k | j = 7 \land k = 4\}$. Here $j = 3 \land k = 4$ is the “precondition”, $j := j + k$ is the statement and $j = 7 \land k = 4$ 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 $\ell_4$, 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 $\ell_4$. 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 $(\ell_j : \phi)_{j \geq 1}$ where $\phi \in W$ (set of well-formed formulas).
4.4 Phase 4: Product of action-tree traces and observational trace.

Given an Action Tree trace $\tau$ and an observational trace $\tau_o$, 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

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

Example 10: Consider the running example. The product of the Action Tree traces ($\tau_1$, $\tau_2$, $\tau_3$) and observational-window trace ($\tau_o$) results the following:

<table>
<thead>
<tr>
<th>$\tau_1 \times \tau_o$</th>
<th>$(\ell_1 : \text{true}, \ell_1 : \text{DB})(\ell_2 : \text{true}, \ell_2 : a' = a + 5)$ $(\ell_3 : \text{true}, \ell_3 : a'' = a' + 1)$ $(\ell_4 : 10 \leq a'' \leq 60, \ell_4 : \text{observe})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_2 \times \tau_o$</td>
<td>$(\ell_1 : \text{true}, \ell_1 : \text{DB})(\ell_2 : \text{true}, \ell_2 : a' = a)$ $(\ell_3 : \text{true}, \ell_3 : a'' = a' + 1)$ $(\ell_4 : 10 \leq a'' \leq 60, \ell_4 : \text{observe})$</td>
</tr>
<tr>
<td>$\tau_3 \times \tau_o$</td>
<td>$(\ell_1 : \text{true}, \ell_1 : \text{DB})(\ell_2 : \text{true}, \ell_2 : a' = a)$ $(\ell_3 : \text{true}, \ell_3 : a'' = a')$ $(\ell_4 : 10 \leq a'' \leq 60, \ell_4 : \text{observe})$</td>
</tr>
</tbody>
</table>

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 $T$. 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:

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

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

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

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

\[ \tau_m^1 = (\ell_1: true, \ell_1: yes)(\ell_2: true, \ell_2: yes)(\ell_3: true, \ell_3: yes)(\ell_4: 10 \leq a' \leq 60, \ell_4: observe) \]

\[ \tau_m^2 = (\ell_1: true, \ell_1: yes)(\ell_2: true, \ell_2: no)(\ell_3: true, \ell_3: yes)(\ell_4: 10 \leq a' \leq 60, \ell_4: observe) \]

\[ \tau_m^3 = (\ell_1: true, \ell_1: yes)(\ell_2: true, \ell_2: no)(\ell_3: true, \ell_3: no)(\ell_4: 10 \leq a' \leq 60, \ell_4: observe) \]

(b) Applying Filter on masked traces

Filter(\( \tau_m^1 \)) = (\( \ell_1: true, \ell_1: yes)(\ell_2: true, \ell_2: yes)(\ell_3: true, \ell_3: yes)(\ell_4: 10 \leq a' \leq 60, \ell_4: observe) \)

Filter(\( \tau_m^2 \)) = (\( \ell_1: true, \ell_1: yes)(\ell_3: true, \ell_3: yes)(\ell_4: 10 \leq a' \leq 60, \ell_4: observe) \)

Filter(\( \tau_m^3 \)) = (\( \ell_1: true, \ell_1: yes)(\ell_4: 10 \leq a' \leq 60, \ell_4: observe) \)

(c) Semantics-based dependencies

<table>
<thead>
<tr>
<th>DEP(Filter(( \tau_m )))</th>
<th>( \ell_3 \rightarrow \ell_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEP(Filter(( \tau_m^2 )))</td>
<td>( \ell_3 \rightarrow \ell_4 )</td>
</tr>
<tr>
<td>DEP(Filter(( \tau_m^3 )))</td>
<td>( \ell_1 \rightarrow \ell_4 )</td>
</tr>
</tbody>
</table>

5 Enhancement using Binary Decision Diagram

The first phase of our proposed algorithm in section 4 results in 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.
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 repetition 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:

<table>
<thead>
<tr>
<th>(\alpha)</th>
<th>(\beta)</th>
<th>(\gamma)</th>
<th>(\Gamma)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The Reduced Binary Decision Diagram of \(\Gamma\) based on the above truth table is shown in Figure 13.

Figure 13: Reduced Binary Decision Diagram of \(\Gamma\)

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

![Ordered Reduced Action Tree Diagram]

For a large Action Tree, these reduction rules reduce 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 \( x \to y \) denotes that an information flows from \( x \) to \( y \). If there is no such path, then there is no information flow.

As the 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: \textit{high} (for private attributes/variables which contain sensitive values) and \textit{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, \textit{i.e.} the security levels of the variables in output statements are \textit{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 \( 8 \xrightarrow{rs} \text{OfferAmt} \rightarrow 11 \) exists even after applying Action-Condition rules. On the other hand, our method removes the false dependence \( 8 \xrightarrow{rs} \text{OfferAmt} \rightarrow 11 \), 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 \( c = \langle 11, \{\text{ID, OfferAmt}\} \rangle \) 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 \( 5 \rightarrow 11 \) 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 \( l \ll n \). Further, time and space complexity can be reduced to linear by applying reduction rules of BDD on Action Tree.
Refining Dependencies for Inf. Flow Analysis of DB Applications

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.executeQuery("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

<table>
<thead>
<tr>
<th>ID</th>
<th>CustName</th>
<th>Address</th>
<th>TransAmt</th>
<th>OfferAmt</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>David</td>
<td>Denver, Colorado</td>
<td>20000</td>
<td>250</td>
<td>Silver</td>
</tr>
<tr>
<td>102</td>
<td>William John</td>
<td>Albany, New York</td>
<td>15000</td>
<td>170</td>
<td>Silver</td>
</tr>
<tr>
<td>103</td>
<td>Rajesh Kumar</td>
<td>Patna, Bihar</td>
<td>24880</td>
<td>190</td>
<td>Gold</td>
</tr>
<tr>
<td>104</td>
<td>Prity Agrawal</td>
<td>Baglore, Karnataka</td>
<td>10000</td>
<td>155</td>
<td>Silver</td>
</tr>
<tr>
<td>105</td>
<td>Jacob Smith</td>
<td>Austin, Texas</td>
<td>12500</td>
<td>165</td>
<td>Silver</td>
</tr>
<tr>
<td>106</td>
<td>Niraj Sinha</td>
<td>Mumbai, Maharatra</td>
<td>2500</td>
<td>00</td>
<td>NULL</td>
</tr>
</tbody>
</table>
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. $l \rightarrow k$. This means that our method results a trace $t = e_1e_2e_3.....e_{n-1}e_n$ such that $DEP(t) = DEP( e_1e_2e_3.....e_{n-1}e_n) = Label(e_{n-1}) \rightarrow Label(e_n) = l \rightarrow k$ where $e_{n-1} = (1: true, 1: yes)$, $e_n = (k: \phi, k: observe)$ and $\phi$ is the observational-window. If we move backward along the phases of the proposed “true” and “yes” in $e_{n-1}$ indicates the following: $\exists t_a \in$ action-tree trace and $\exists t_o \in$ observational-trace such that $\pi_l(t_a) \times \pi_l(t_o) = (\phi_a, A_a) \times (\phi_o) = (1: true, 1: A_\phi)$, where $\pi_l$ is the projection operation of $l^{th}$ elements from the trace, and $\phi_a \land \phi_o \neq \emptyset$. For $j > l$, $\pi_l(t_a) \times \pi_l(t_o) = (x, y)$ 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 $\phi_a \land \phi_o \neq \emptyset$ 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


Refining Dependencies for Inf. Flow Analysis of DB Applications


