Computational Program Dependence Graph and its Application to Information Flow Security

Chunyan Mu
Computational Program Dependence Graph and its Application to Information Flow Security

C. Mu

Abstract

This paper develops a novel approach that analyses dependencies of programs in a quantitative aspect. We introduce a definition of Quantitative Program Dependence Graph (QPDG) which can be used to model a program's behaviour given spaces of inputs. The programs we consider are in a core while-language. We also present the semantics for the purpose of building QPDGs. The QPDG reasons about the programs' quantitative uncertainty behaviours based on a probabilistic analysis. It can be used to characterise dependence analysis of programs in a quantitative way. We next provides a further optimisation on the QPDG by doing slicing in order to perform an information flow analysis, e.g. how input variables at the source node might affect a given output variable at the target node and how much. Finally, we suggest its application to quantified secure information flow analysis for programs, and show that simple and intuitive computation can be obtained.
Abstract

This paper develops a novel approach that analyses dependencies of programs in a quantitative aspect. We introduce a definition of Quantitative Program Dependence Graph (QPDG) which can be used to model a program's behaviour given spaces of inputs. The programs we consider are in a core while-language. We also present the semantics for the purpose of building QPDGs. The QPDG reasons about the programs' quantitative uncertainty behaviours based on a probabilistic analysis. It can be used to characterise dependence analysis of programs in a quantitative way. We next provides a further optimisation on the QPDG by doing slicing in order to perform an information flow analysis, e.g. how input variables at the source node might affect a given output variable at the target node and how much. Finally, we suggest its application to quantified secure information flow analysis for programs, and show that simple and intuitive computation can be obtained.

About the authors

Chunyan Mu received her PhD from King's College London, and is currently a temporary lecture of Computing Science at Newcastle University. Her research interests include: language-based security, programming languages, information flow security.

Suggested keywords

LANGUAGE
DEPENDENCE
MEASUREMENT
GRAPH
INFORMATION
FLOW
SECURITY
Computational Program Dependence Graph and its Application to Information Flow Security

Chunyan Mu

School of Computing Science
Newcastle University NE1 7RU
chunyan.mu@ncl.ac.uk

Abstract

This paper develops a novel approach that analyses dependencies of programs in a quantitative aspect. We introduce a definition of Quantitative Program Dependence Graph (QPDG) which can be used to model a program’s behaviour given spaces of inputs. The programs we consider are in a core while-language. We also present the semantics for the purpose of building QPDGs. The QPDG reasons about the programs’ quantitative uncertainty behaviours based on a probabilistic analysis. It can be used to characterise dependence analysis of programs in a quantitative way. We next provides a further optimisation on the QPDG by doing slicing in order to perform an information flow analysis, e.g. how input variables at the source node might affect a given output variable at the target node and how much. Finally, we suggest its application to quantified information flow analysis for programs, and show that simple and intuitive computation can be obtained.

Keywords: Language, Dependence, Measurement, Graph, Information, flow, Security

1 Introduction

Program executions contain many dependencies. Dependence analysis between entities is an important part of program analysis. Although the dependence decision made at two entities of interest is normally treated as a binary decision, the information obtained during the dependence analysis can be quantified.

A quantitative analysis of program execution is essential to the system design process and information flow control mechanisms. A quantitative dependence analysis can be used to characterise the exact nature of dependencies between statements or variables for program analysis. In this paper, we propose a representation of the program dependence graph that can be used to capture the properties of information flow between variables by executing the program. By using the idea of information theory [18], we compute the quantity of the interference introduced by data dependence information among variables (which can be with different security
levels) at nodes of interest. Specifically, we propose to define and build program dependence graphs which leads to a simple algorithm for providing efficient quantified information flow analysis. As an application, it can be used to provide information flow analysis for programs and to measure the information flow between two components and thus provide an approach in information flow control mechanisms given a threshold. In the security community, it can be used to capture the information flow from high-level private inputs to low-level public outputs by executing the program, and therefore we can calculate how much information can be learned about the high security input operation by observing the low-level security output.

The main contributions of this paper are therefore summarised as follows. We introduce a definition of Quantitative Program Dependence Graph (QPDG), which can be used to provide a quantitative analysis of dependencies between program variables by executing the program. We consider a simple programming language, give the syntax and semantics of the language, describe the QPDG representation for this language, and construct the dependencies between variables from the syntax of the program statements. By semantically interpreting the concepts of control and data dependence, we derive a denotational definition of the control and data demand generated by variables at program points of interests. We then perform a quantitative analysis on the QPDG semantics and suggest a method to measure the dependence between program variables at points of interest by using Shannon’s information theory. Intuitively, we are particularly interested in which variables at a source node might affect a given variable at a target node and how much in information bits (uncertainty measurement). For this purpose, we adapt the PDG nodes to accommodate the stores and dependencies between variables introduced by the statement. Furthermore, we give a simple algorithm in order to build a reduced QPDG which can be used to produce more effective analysis of quantified dependence and information flow. We also discuss the possible applications of the reduced QPDG to information flow measurement in the security community.

The rest of the paper is organised as follows. Section 2 briefly goes over the definitions of PDG, random variables and information entropy. Section 3 defines the syntax and semantics for a simple while language and semantically describes the QPDG for this language. In Section 4, we present an algorithm for building a reduced QPDG to provide a more efficient quantitative dependence analysis and discuss a possible application to information flow measurement for security. The final Section summarises our work.

2 Preliminary

In this section, we briefly review some definitions in the relevant background including program dependence graph, random variables and program, and information theory.
2.1 Program Dependence Graph

Many analyses and transformations of programs are based on dependence relations which are normally represented by program dependence graph (PDG) [6]. The PDG plays an important role in expressing the essential dependencies of atomic program operations. We use the dependence graph as our intermediate representation basis.

Intuitively, the dependence graph can be viewed as a data structure in which edges represent dependencies between operations. Dependence graphs integrate data and control dependence information into a single structure, making efficient algorithms for program analysis. Typically, a PDG has two types of dependence edges: a data-dependence edge and a control-dependence edge. A data-dependence edge from statement $s_1$ to statement $s_2$ means that the computation performed in $s_2$ depends on the value computed in $s_1$. A control-dependence edge from $s_1$ to $s_2$ implies that $s_2$ may or may not be executed depending on the boolean outcome of $s_1$, for instance, a if-statement or while-loop. For every dependence edge in the data dependence graph, there is a corresponding path in the program dependence graph.

A dependence graph consists of set of nodes representing functional operators and a set of edges representing the dependencies and precedence relations that exist between those operations. If there is a dependence between two statements in the source program, there must be a path between the corresponding nodes in the dependence graph.

In this paper, we propose to develop a novel representation of PDG that can be used to capture the quantified dependencies between variables rather than just the statements themselves. For instance, we are interested in how a specific output variable at a target node might be affected by a given input variable at a source node.

2.2 Shannon’s Measure of Entropy

Information theory introduced the definition of entropy, $H$, to measure the average uncertainty in random variables. Shannon’s measures were based on a logarithmic measure of the unexpectedness in a probabilistic event (random variable). The unexpectedness of an event which occurred with some non-zero probability $p$ was $\log_2 \frac{1}{p}$. Hence the total information carried by a set of $n$ events was computed as the weighted sum of their unexpectedness:

$$\mathcal{H} = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i}$$

where $p_i \log_2 \frac{1}{p_i} = 0$ if $p_i = 0$. This quantity is called the entropy of the set of events.

Considering a program as a state transformer, random variable $X$ is a mapping between two states which are equipped with distributions. We generally use the language of random variables rather than talk directly about distributions. Let $p(x)$ denote the probability that $X$ takes the value $x$, then the entropy $\mathcal{H}(X)$ of
discrete random variable $X$ is defined as:

$$\mathcal{H}(X) = \sum_x p(x) \log_2 \frac{1}{p(x)} = -\sum_x p(x) \log_2 p(x)$$

The entropy is measured in bits and is a measure of the uncertainty of a random variable, which can never be negative. Intuitively, it is the number of bits on the average required to describe the random variable. Furthermore, given two random variables $X$ and $Y$, the notion of conditional entropy $\mathcal{H}(Y|X) = \sum_x p(x) \mathcal{H}(Y|X = x)$ suggests possible dependencies between random variables, i.e. knowledge of one may change the information about the other. The concept of mutual information is a measure of the amount of information that one random variable contains about another one, i.e. shared information. It implies the reduction in the uncertainty of one random variable due to the knowledge of the other. Let $p(x, y)$ denote the joint distribution of $x \in X$ and $y \in Y$, the notion of mutual information between $X$ and $Y$, $\mathcal{I}(X;Y)$, is given by:

$$\mathcal{I}(X;Y) = \sum_x \sum_y p(x, y) \log_2 \frac{p(x, y)}{p(x)p(y)}$$

$$= \mathcal{H}(X) - \mathcal{H}(X|Y) = \mathcal{H}(Y) - \mathcal{H}(Y|X)$$

There are also conditional versions of mutual information: $\mathcal{I}(X;Y|Z)$ denotes the mutual information between $X$ and $Y$ given the knowledge of $Z$. The conditional mutual information can be considered as the reduction in the uncertainty of $X$ due to knowledge of $Y$ when $Z$ is given:

$$\mathcal{I}(X;Y|Z) = \mathcal{H}(X|Z) + \mathcal{H}(Y|Z) - \mathcal{H}(X,Y|Z)$$

$$= \mathcal{H}(X|Z) - \mathcal{H}(X|Y,Z)$$

$$= \mathcal{H}(Y|Z) - \mathcal{H}(Y|X,Z)$$

In this paper, we consider program variables as random variables and define the quantity of dependencies or information flow between program variables by using the concept of conditional mutual information.

3 The language and program dependence

This section presents the language and semantics to semantically express the dependencies between program variables within program operations. In order to provide a quantitative analysis of program dependencies between variables for programs, we consider the program variables as random variables which take probability distributions as mapped values rather than single values, i.e. there are probability distributions on the state space rather than just values.

3.1 The language

To simplify our presentation and focus our attention on the problem of quantified dependence analysis of programs, we consider a simplified language.
The abstract syntax

We present the syntax of the core language in Table 1.

<table>
<thead>
<tr>
<th>Class</th>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stmt</td>
<td>s</td>
<td>::= c1 c ::= (input(x))(skip)(x := e)(s1; s2)(if b then s else s)(while b do s)</td>
</tr>
<tr>
<td>Lab</td>
<td>l</td>
<td>∈ Lab</td>
</tr>
<tr>
<td>Ide</td>
<td>x</td>
<td>∈ Ide</td>
</tr>
<tr>
<td>Exp</td>
<td>e</td>
<td>::= x</td>
</tr>
<tr>
<td>BExp</td>
<td>b</td>
<td>::= ¬b</td>
</tr>
</tbody>
</table>

Table 1
The abstract syntax of the language

where s ranges over statements, l denotes a label, x ranges over a set of variables, b ranges over a set of boolean expressions, and e ranges over a set of arithmetic expressions.

Semantic domains

\[ \text{Val} \overset{df}{=} \mu \quad \sigma \in \text{Store} \overset{df}{=} \text{Ide} \rightarrow \text{Val} \]

\[ \delta \overset{df}{=} \mathcal{P}(\text{Ide}) \quad \Delta_l \in \text{Dep} \overset{df}{=} \mathcal{P}(\text{Ide} \rightarrow \delta \cup \text{Val}) \]

where Val is a probability space µ; σ ∈ Store is a state which maps Ide to probability spaces; δ denotes a set of program variables (a given expression evaluation depends on); Δl denotes a superset of the dependencies between variables for a statement with label l, each of which can be treated as a function of type x → δ ∪ Val where x ∈ Ide: variable x depends on a set of variables in δ or Val for (random inputs) by executing the statement with label l.

The semantics

Denotational semantics is closely related to dependence analysis. We want to reformulate the probabilistic denotational semantics [11] so that the dependencies can be extracted and recorded for the program. The assignment is ignored unless it affects the visible contents of the program store. A definition of our denotational semantics is given in Table 2.

An arithmetic expression Exp is a function \([e] : \sigma \rightarrow \text{Val}\) by using two’s-complement interpretations of +, −, *, / as standard. Let δ(e) denote a set of variables on which evaluating e depends, i.e. the value of \([e]\) depends at most on the variables in δ(e). A boolean expression BExp is interpreted as a function \([b] : \sigma \rightarrow \sigma\). The function defines the part of the store matched by the condition b. In addition, the value of \([b]\) depends at most on the variables in δ(b). For example, assume the current store is

\[ \sigma(x \mapsto [0 \rightarrow \frac{1}{4}, 1 \rightarrow \frac{1}{4}, 2 \rightarrow \frac{1}{4}, 3 \rightarrow \frac{1}{4}]) \]

and consider boolean expression b is \([x > 1]\), we have δ(b) = \{x\} and the updated store under condition b as:

\[ \sigma(x \mapsto [2 \rightarrow \frac{1}{4}, 3 \rightarrow \frac{1}{4}]) \]
Table 2
Denotational Semantics of Programs

\[
\begin{align*}
[s] & : \text{Stmt} \rightarrow (\text{Store} \rightarrow \text{Store}) \rightarrow (\Delta \rightarrow \Delta) \\
[e] & : \text{Exp} \rightarrow (\text{Store} \rightarrow \text{Val}) \rightarrow \delta \\
[b] & : \text{BExp} \rightarrow (\text{Store} \rightarrow \text{Store}) \rightarrow \delta \\
\end{align*}
\]

\[
\begin{align*}
\text{input}(x) & : \sigma_l \Delta_l \overset{df}{=} \sigma_l(x \Rightarrow \mu_x) \Delta_l(x \Rightarrow \mu_x) \\
x := e & : \sigma_l \Delta_l \overset{df}{=} \lambda W, \sigma_l([x := e]_l^{-1}(W)) \Delta_l(x \Rightarrow \delta(e)) \\
[s_1; s_2] & : \sigma_l \Delta_l \overset{df}{=} \delta [s_2] \circ [s_1] \sigma_l \Delta_l \\
\text{ifb then } s_1 \text{ else } s_2 & : \sigma_l \Delta_l \overset{df}{=} \Delta_l([x \rightarrow \delta_x \cup \delta(b)|x \in \text{Def}(s_1)] [s_1] \circ \sigma_l \Delta_l \cup \\
& \Delta_l([x \rightarrow \delta_x \cup \delta(b)|x \in \text{Def}(s_2)] [s_2] \circ \neg b \sigma_l \Delta_l \\
\text{while } b \text{ do } s & : \sigma_l \Delta_l \overset{df}{=} \Delta_l([x \rightarrow \delta_x \cup \delta(b)|x \in \text{Def}(s)]) \\
& \neg b [\lim_{n \to \infty} (\lambda \sigma_l^{n} \Delta_l \sigma_l \Delta_l \cup \delta \{s\} \circ \delta \Delta_l^{n})] (\lambda \sigma_l \Delta_l \Delta_l) \\
\text{where, } \neg b \sigma_l & = \lambda X, \sigma_l (X \cap \mathcal{M}_b) \\
\end{align*}
\]

The semantics is a store transformer carried program dependencies. The meaning of a statement \(s\) is not a transformer of the entire input store \(\sigma\) but a transformer only of the part of the store consisting of the variables that are defined in \(s\). In addition, the store is a map from the variables \(\text{IDE}\) to a probability space \(\mu\) rather than from variables to their values. For a given \(\text{Stmt}\ s\), \(\text{Def}(s)\) returns the superset of all variables that may be assigned to. Given a label \(l\), \(\Delta_l\) returns the superset of maps: \(\mathcal{P}(\text{IDE} \rightarrow \delta \cup \text{Val})\) which constitute dependencies for the \(\text{Stmt}\) with that label. Specifically,

- **Random input** \(\text{input}(x)\) assigns a probability space \(\mu_x\) to \(x\), and builds a dependence relation such that variable \(x\) depends on the input space: \(\text{Val}_x = \mu_x\), \(\Delta_l(x \rightarrow \text{Val}_x)\).
- **Assignment** updates the store such that the state of assigned variable \(x\) is updated to become that of expression \(e\), and updates the dependence such that \(x\) depends on \(\delta(e)\): \(\Delta(x \rightarrow \delta(e))\). The distribution transformation function for assignment is presented by using an inverse image: \(\lambda W, \sigma_l([x := e]_l^{-1}(W))\) to keep the measurability of the semantic function [11]. For all measurable \(W \in \sigma', [x := e]_l^{-1}(W)\) is measurable in \(\sigma\), where \(\sigma\) and \(\sigma'\) denote the state before and after the assignment operator.
- The distribution transformation function for the **sequential composition operator** is obtained via functional composition:

\[
[s_1; s_2] \sigma_l \Delta_l = [s_2] \circ [s_1] \sigma_l \Delta_l
\]

Note that \(\Delta\) is a binary relation on variables. The relational composition is considered as follows: assume \(\delta_x\) denotes the set of variables that \(x\) depends on by executing the statement \(s_1\), \(\delta_y\) denotes the set of variables \(y\) depends on by
executing the statement $s_2$, assume $x \in \text{Def}(s_1), y \in \text{Def}(s_2)$

$$[s_1] \Delta_l = \Delta_l'(\{x \to \delta_x \mid x \in \text{Def}(s_1)\})$$

$$[s_2] \Delta_l' = \Delta_l'(\{x \to \delta_x \mid x \in \text{Def}(s_1) \land x \notin \text{Def}(s_2)\}) \cup \Delta_l''(\{y \to (\lambda x. \delta_y) \delta_x \mid y \in \text{Def}(s_2), x \in \text{Def}(s_1)\})$$

where we use $(\lambda x. \delta_y) \delta_x = [\delta_x/x] \delta_y$ to indicate that all occurrences of $x$ in $\delta_y$ are substituted by $\delta_x$. We therefore write that,

$$[s_1; s_2] \Delta_l = ([s_2] \circ [s_1]) \Delta_l$$

$$= [s_2] \Delta_l'(\{x \to \delta_x \mid x \in \text{Def}(s_1)\})$$

$$= \Delta_l'(\{x \to \delta_x \mid x \in \text{Def}(s_1) \land x \notin \text{Def}(s_2)\}) \cup \Delta_l''(\{y \to (\lambda x. \delta_y) \delta_x \mid y \in \text{Def}(s_2), x \in \text{Def}(s_1)\})$$

- The boolean function $[b]$ for boolean test $b$ defines the part of the space defined in the current store matched by the condition $b$: $[b] \sigma = \lambda X. \sigma(X \cap M_b)$. The boolean test causes the space to split apart, $X \cap M_b$ denotes the part of the space which leads boolean test $b$ to be true. We use $\delta(b)$ to denote the set of variables that valuating $b$ depends on.

- A Conditional statement is executed on the conditional probability distributions for either the true branch or false branch:

$$\Delta_l(\{x \to \delta_x \cup \delta(b) \mid x \in \text{Def}(s_1)\})[s_1] \circ [b] \sigma_l \Delta_l \bigcup \Delta_l(\{x \to \delta_x \cup \delta(b) \mid x \in \text{Def}(s_2)\})[s_2] \circ [-b] \sigma_l \Delta_l$$

where $\Delta_l(\{x \to \delta_x \cup \delta(b) \mid x \in \text{Def}(s_1)\})$ means that, $\forall x \in \text{Def}(s_i)$ and $i = 1, 2$, we update the dependence relation of $x$ by union $\delta(b)$, where $\text{Def}(s_i)$ denotes a set of variables are assigned to during evaluation of the statement $s_i$, $\delta_x$ denotes the set of variables that $x$ depends on in $\Delta_l$ after executing $[s_i]$, i.e. $[s_i] \circ [b] \Delta_l$. For instance, consider a piece of program with an if statement:

$$[\text{if } (x > 1) \text{ then } y := y - 1 \text{ else } y := y + 1:]_1$$

Assume the space of $x, y$ at the beginning of the program is written as:

$$\sigma_l[x \mapsto (0 \to \frac{1}{4} \quad 1 \to \frac{3}{4} \quad 2 \to \frac{1}{4} \quad 3 \to \frac{1}{4}), \ y \mapsto (0 \to 1)]$$

After executing the if statement, according to the semantics, the store is obviously updated as:

$$\sigma_l[x \mapsto \begin{pmatrix} 0 \to \frac{1}{4} & 1 \to \frac{1}{4} \\ 2 \to \frac{1}{4} & 3 \to \frac{1}{4} \end{pmatrix}, \ y \mapsto \begin{pmatrix} -1 \to \frac{1}{2} \\ 1 \to \frac{1}{2} \end{pmatrix}]$$

and the dependence is updated as:

$$\Delta_l(y \to y \cup \delta(b)) = \Delta_l(y \to \{x, y\})$$

i.e.

$$[\text{if } (x > 1) \text{ then } y := y - 1 \text{ else } y := y + 1:]_1 \sigma_l \Delta_l$$
\[ \sigma_l[x \mapsto \begin{pmatrix} 0 & 1 & 2 \\ 1 & 3 & 4 \end{pmatrix}, y \mapsto \begin{pmatrix} -1 & 1 \\ 2 & 4 \end{pmatrix}] \Delta_l(y \to \{x, y\}) \]

- In the while loop, the state space with distribution \( \mu \) defined in the current store goes around the loop, and at each iteration, the remaining part that makes test \( b \) false breaks off and exits the loop, while the rest of the space goes around again. The output distribution \( \{\text{while } b \text{ do } s\} \sigma_l \) is thus the sum of all the partitions that finally find their way out. In this paper, we only consider terminating loops. In addition, the dependence representation of while loops is similar to that of the if statement, \( \{x \to \delta_x \cup \delta(b) | x \in \text{DEF}(s)\} \) means that the dependencies introduced by the loop: the depending variables of the defined variables in the loop body are updated by union \( \delta(b) \).

### 3.2 Quantitative program dependence graph

A program dependence graph defines a partial ordering on the statements and the program performance to preserve the semantics of the original program. For our purpose of quantitative analysis on program dependencies, we introduce a definition of quantitative program dependence graph based on the semantics defined in Section 3.1. A quantified program dependence graph consists of a set of nodes representing atomic computations (assignments and predicate expressions) with current stores regarding to \( \sigma \) and a set of edges representing the dependencies regarding to \( \Delta \).

There are some differences between our QPDG and the standard PDG.

- Rather than considering dependencies between statements, we consider the dependencies between variables.
- The node of the QPDG is labelled by program statement blocks (which can be assignments, if statements, while loops): \( \text{Block} \in \mathcal{P}(	ext{STMT}) \). Each node accommodates the stores which records the states of each variable obtained from the stores \( \sigma \) and the dependence relations defined in \( \Delta \) given by the semantic functions. Intuitively, the node of the QPDG can be viewed as a transformation box, which accommodates a statement block with label \( l \), a set of entry ports and a set of exit ports, and a set of directed internal edges between entry ports and exit ports of the node. Each entry port stores the state of a “will-be-used” variable within the block before executing the statement block. Each exit port stores the state of each defined variable within the node after executing the statement block.
- Directed internal edges between entry ports and exit ports of a node are used for connecting multiple pairs of variables according to their dependencies obtained from \( \Delta \) within the node. The dependence (denoted by an internal edge) between each pair is introduced by executing the statement block accommodated at this node. We give a definition of the quantity of the internal dependence edge based on conditional mutual information which will be discussed later.
- External dependence edges are essentially data dependence edges between two nodes. It starts from an exit port (accommodates a defined variable and its
state) of a node and ends at an entry port (accommodates a used variable and its states) of another node, and denotes that there is a precedence relation and a data dependence relation between these two variables at these two nodes.

Example 3.1  Consider a piece of program

\[
\begin{align*}
\text{input}(x) &; \\
y &:= 0 l_1 \\
\text{if } (x \mod 2 == 0) & \text{ then } y := y - 1 \text{ else } y := y + 1 \ l_3
\end{align*}
\]

Assume the state of \( x \) is \( \sigma_{l_1} [x \mapsto \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}] \) given by \( \text{input}(x) \). We present the QPDG of this example in Fig. 1 to show some intuition of the graph.

We present the elements of the graph for this example in Table 3.

<table>
<thead>
<tr>
<th>semantic blocks</th>
<th>entry ports</th>
<th>exit ports</th>
<th>dependencies ( \Delta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{input}(x) )</td>
<td>( \text{VAL}_x )</td>
<td>( x \mapsto \begin{pmatrix} 0 &amp; 1 \ 2 &amp; 3 \end{pmatrix} )</td>
<td>( x \rightarrow \text{VAL}_x )</td>
</tr>
<tr>
<td>( y := 0 )</td>
<td>( \text{VAL}_y )</td>
<td>( y \mapsto (0 \rightarrow 1) )</td>
<td>( y \rightarrow \text{VAL}_y )</td>
</tr>
<tr>
<td>( \text{if } (x % 2 == 0) \text{ then } y := y - 1 \text{ else } y := y + 1 )</td>
<td>( x \mapsto \begin{pmatrix} 0 &amp; 1 \ 2 &amp; 3 \end{pmatrix} )</td>
<td>( y \mapsto (\neg 1 \rightarrow 1 \rightarrow 1) )</td>
<td>( y \rightarrow {x, y} )</td>
</tr>
</tbody>
</table>

Table 3  A simple example of QPDG: the nodes

Intuitively, each node is a transformation box with two sides of “ports”: entry ports and exit ports. Each entry port accommodates a “will-be-used” variable with its state before executing the statement block. Each exit port accommodates a “defined variable” with its state after executing the statement block. If there is an internal connection between at an entry port and an exit port within the box then
there is a dependence between them which is introduced by the statement of this node. For instance, the internal dependence edges with the node of if statement means the definition of $y$ in the if statement depends on variables $x$ and $y$ according to the semantics. Let $q(\Delta_{b_t}(y \to x))$ denote the quantity of the dependence between the variable $y$ and the variable $x$, then

$$q(\Delta_{b_t}(y \to x)) = I(Y'; X|Y) = I(Y'; X) = H(Y') = 1$$

where $X, Y, Y'$ denote the random variables of the variable $x$ at entry port before executing the program, program variable $y$ at the entry port (before executing the program), and program variable $y$ at the exit port (after executing the program).

In addition, the external data dependence edge between the node of $\text{input}(x)$ and the node of if statement suggests the used variable $x$ at the if statement node (entry port) depends on the defined variable at the node of $\text{input}(x)$ (exit port).

We now present the definition of quantitative program dependence graph as follows.

**Definition 3.2** Quantitative program dependence graph (QPDG). A QPDG is defined as a pair:

$$QPDG = (N, E_x)$$

where,

- $N = \mathcal{P}(\text{Stmt}, P_{entry}, P_{exit}, E_i)$ is a power set of tuples of executable statement boxes each of which accommodates a statement block (which can be an assignment, if statement, or while loop), a set of entry ports $P_{entry}$ and a set of exit ports $P_{exit}$, and directed internal dependencies edges denoted by $E_i$.
  - Each entry port accommodates a used variable with its the state defined in $\sigma$ before executing the statement block.
  - Each exit port accommodates a defined variable with its state of each variable after executing the statement block.
  - Directed internal edges $E_i$ between entry ports and exit ports connects multiple pairs of variables which can be extracted from their dependencies defined in $\Delta$ by the semantics. Specifically, $E_i$ is a set of directed edges inside the nodes. Each edge $e \in E_i$ is a tuple $(p_{entry}, p_{exit}, q_e)$, where $p_{entry} \in P_{entry}$, $p_{exit} \in P_{exit}$, and $q_e$ denotes the quantity of $e$. Let $X$ denote the random variable locates at $P_{entry}$, and $Y'$ be the random variable locates at $P_{exit}$, and $Z$ denote the joint random variable of a set of variables locate in $Q_{entry} = P_{entry}/P_{entry}$, and there exists an internal edge $e' \in E_i$, $q_{entry} \in Q_{entry}$, and $e' \neq e$ where $e' = (q_{entry}, p_{exit}, q_{e'})$. Intuitively, each internal edge connects an entry port which accommodates a variable used in the node and an exit port which accommodates a defined variable in the node. Note that the internal data flow edges $E_i$ implies the dependencies introduced by the statement block of the nodes and the node can be nested regarding to the nested if statements and while loops.

- $E_x$ is a set of directed external edges among the nodes. Each edge of $E_x$ starts from an exit port of one node $m$ and ends at an entry port of another node $n$. Specifically, a defined variable at node $m$ is used at node $n$, where $m, n \in N$. 

Intuitively, a data flow edge between nodes actually connects programs variables and implies a data flow dependence.

**Example 3.3** Consider an example program $P$ and its QPDG in Fig. 2.

11. `input(x);`
12. `z := 1;`
13. `y := 0;`
14. `while (x > y) {
    y := y + 1;
    z := z * y;
}`
15. `print(y);`

Assume the probability distribution of possible values of $x$ produced by statement `input(x)` is $\mu_x: x \mapsto \begin{pmatrix} 0 \rightarrow \frac{1}{4} & 1 \rightarrow \frac{1}{4} \\ 2 \rightarrow \frac{1}{4} & 3 \rightarrow \frac{1}{4} \end{pmatrix}$. According to semantic function defined in Table 2, the transformation of program $P_1$ can be written as:

\[
\begin{align*}
\sigma_{l_1} \Delta_{l_1} &= \sigma_{l_1} [x \mapsto \begin{pmatrix} 0 \rightarrow \frac{1}{4} & 1 \rightarrow \frac{1}{4} \\ 2 \rightarrow \frac{1}{4} & 3 \rightarrow \frac{1}{4} \end{pmatrix}] \Delta_{l_1} (x \rightarrow \text{VAL}_x) \\
\sigma_{l_2} \Delta_{l_2} &= \sigma_{l_2} [x \mapsto \begin{pmatrix} 0 \rightarrow \frac{1}{4} & 1 \rightarrow \frac{1}{4} \\ 2 \rightarrow \frac{1}{4} & 3 \rightarrow \frac{1}{4} \end{pmatrix}] , z \mapsto 1 \rightarrow 1] \Delta_{l_2} (z \rightarrow \text{VAL}_z) \\
\sigma_{l_3} \Delta_{l_3} &= \sigma_{l_3} [x \mapsto \begin{pmatrix} 0 \rightarrow \frac{1}{4} & 1 \rightarrow \frac{1}{4} \\ 2 \rightarrow \frac{1}{4} & 3 \rightarrow \frac{1}{4} \end{pmatrix}] , z \mapsto 1 \rightarrow 1 , y \mapsto 0 \rightarrow 1] \Delta_{l_3} (y \rightarrow \text{VAL}_y) \\
\sigma_{l_4} \Delta_{l_4} &= \sigma_{l_4} [x \mapsto \begin{pmatrix} 0 \rightarrow \frac{1}{4} & 1 \rightarrow \frac{1}{4} \\ 2 \rightarrow \frac{1}{4} & 3 \rightarrow \frac{1}{4} \end{pmatrix}] , z \mapsto 1 \rightarrow 1 , y \mapsto 0 \rightarrow 1] \Delta_{l_4} (y \rightarrow \text{VAL}_y) \\
\end{align*}
\]

A translation of the $P$ into the QPDG is presented in the right part of Fig. 2. Let us consider the node of while loop denoted as $n_{l_4}$ as an example. Obviously,

\[
n_{l_4} = (\{\text{while } (x > y) \text{ do } y := y + 1; z := z * y; \} |_{l_4} , P_{\text{entry}}, P_{\text{exit}}, \{e_i | 1 \leq i \leq 4\})
\]

where, $P_{\text{entry}} = \{x, z, y\}$, $P_{\text{exit}} = \{x, z, y\}$. Consider two internal dependence edges of interest for an example:

\[
e_1 = (x, z, \frac{2}{3}) , e_2 = (x, y, 2)
\]

Note that the quantity of internal dependence edge $e_i$ is computed by conditional mutual information as discussed above. For example, the quantity of the dependence
edge $e_1$ and $e_2$ of interests can be computed by:

\[ q_{e_1} = I(Z'; X|Y) = I(Z'; X) = \frac{2}{3} \]
\[ q_{e_2} = I(Y'; X|Y) = I(Y'; X) = 2 \]

where $X, Y, Y', Z, Z'$ denote the random variables of the program variable $x$ before executing the loop block, program variable $y$ before and after executing the loop block, program variable $z$ before and after executing the loop block.

4 Reducing QPDG by Backward Slicing for information flow analysis

The dependence we concern for information flow analysis relates a variable at one program point to a variable at another, e.g. we are particularly interested in whether or not the output variable at target node depends on the input variable at source node and how much. In this section, we provide a reduction on the QPDG by doing slicing [20] on it in order to extract the part of a program (the slice) which is relevant to a subset of the program dependence behaviour of interest. Given the target output node and source input node of interest, we perform a backward slicing to chop out the nodes and data flow components without affecting the execution of the target node. In this way, we chop the program dependence graph to extract the subgraph of the program that affects the values of the target node and filters the components of the graphs that do not affect the values of it. Given a source node, if the source node is not included in the chopped graph, the target node is not affected by the source node and hence there is no dependence relation between the output variable at the target node and the input variable defined at the source node. If both of them are included in the reduced QPDG then the target node depends on the source node, the quantity of dependence is defined as the conditional mutual information between the output variable at the target node (assuming that the relevant random variable is denoted by $Y'$) and the defined variable at the source node (assuming that the relevant random variable is denoted by $X$) on the condition of the joint random variables of other inputs (which affect the target node but except the input variable at source node) denoted by $Z$ that $Y'$ depend on: $I(Y'; X|Z)$.

4.1 Reducing QPDG

In order to consider the quantified dependence analysis between a specific input variable and a specific output variable and information flow analysis, we reduce the QPDG based on backward slicing, i.e. we filter the nodes that are not affecting the output variable at the target point. A backward slice consists of all the nodes that affect a given target point in the program. In what follows, we present the description to build a reduced QPDG for our purpose of quantifying dependencies and information flow analysis.

(i) Given the observed output/observation variable and its output point (the target node of interest), and the sensitive input variable and its input point (the
source node of interest). Assume the original quantitative program dependence graph obtained by the semantics is denoted by $G_0$.

(ii) Doing a standard backward slicing from the observed (output) variable at the target point: obtain the sets of statements of the program that influence the value of the observed variable; find all the variables at their definition point (denoted by $Z_0 = \{(z, n) | n \in N, z \in \text{IDE}\})$ which affect the final value of the output variable at the target point: a definition of variable $z$ at node $n$ reaches the target node. Let $G_1$ denote the graph obtained after doing backward slice.

(iii) If the source node is not included in the obtained graph $G_1$, then there is no dependence or interference between the the definition of the input variable at the source node and the observed output variable at the target node. The algorithm is terminated here and the quantity of the dependence between the input variable at source and the output variable at target is 0.

(iv) Otherwise, we compute the quantity of the dependence denoted by $d$ between the relevant random variable defined at source node denoted by $X$ and the relevant random variable used at the target node denoted by $Y'$:

$$d = I(Y'; X|Z)$$

where $Z$ denote the joint random variable of $Z_0/X$, i.e. $Z = \{(z, n) | n \in N, (z, n) \in Z_0 \land z \neq x\}$.

4.2 Application to information flow security

Information-flow security enforces limits on the use of information propagation in programs. The goal of information flow security is to ensure that the information propagates throughout the execution environment without security violations so that no secure information is leaked to public outputs. Absolute information-flow properties, such as non-interference [7], are too restrictive and rarely satisfied by real programs. One promising approach to relaxing non-interference is to quantify the information flow and to determine how much information is being leaked, thus allowing us to tolerate “small” leaks. It is desirable to have a quantitative analysis of the programs to judge whether they are secure or not on the basis of the quantity of confidential information deducible by the public. Quantitative information flow (QIF) is concerned with measuring the amount of information flow caused by interference between variables and therefore relaxes the well known non-interference [7] property by introducing a new policy: the program is secure if the amount of information flow from confidential inputs to public outputs is not too big.

Assume we have two types of input variables: $H$ (confidential or sensitive) and $L$ (public). High variables contain confidential information when the program is run. Low variables do not contain confidential information before the program is run. The attacker therefore intends to learn about some of the confidential inputs $H$ by observing the series of low variable outputs $L$. The confidential information is leaked to the attacker by examining the interference between the high-level variables and low-level variables during the execution of the program. We need to verify that the
final values of variables in $L$ give no information about the initial values of variables in $H$. We are also interested in the degree of such interference or information flow, i.e. how much the high-level components interferes with or flows to the low-level components.

We argue that the reduced QPDG discussed in this paper can be used to provide a quantified information flow analysis. The chopped QPDG by doing combined backward slicing can be used to extract and capture the secure information flow within the programs. By building a reduced QPDG, we provide a more efficient quantified leakage analyser: the statements of the program which do not contribute to bring secure information flows from high input to low output are filtered. Intuitively, given two points of interest (high input and low output), the chopped QPDG consists of those statements that can transmit a change from the source (high inputs) to the target (low outputs).

**Example 4.1** Let us look back the program presented in Example 3.3. Assume $x$ is high level sensitive variable, $y$ and $z$ are low level variables. Assume the observed output variable is $y$. The source node of interest is statement `input(x);`, and the target node of interest is the output of $y$. We present the reduced QPDG in Fig. 3. Note that the statements not affecting the target node have been filtered.

![Fig. 3. Leakage analysis on reduced QPDG](image)

The information flow from high input to low output can be calculated as the quantity of the dependence between them:

$$d = I(Y'; X | Y) = I(Y'; X) = H(Y') = 2$$

Obviously, if the quantity of the dependence is 0, there is no information flow from high input to low output, i.e. the program satisfies non-interference and is secure. Specifically, the analysed program $S_l$ is considered secure if $\forall x \in L$, there is no $y \in H$ such that $\Delta_l(x \rightarrow y)$. 

14
5 Related Work

There are a number of techniques have been developed for dependence representations of programs, *e.g.* [6,17,19,9] etc. They are difficult to automatically capture quantified dependence between variables for our purpose of computational information flow analysis. Quantitative information flow (QIF) has recently become an active research topic, *e.g.* [2,4,12,10,13,1,8,3]. As a precursor of this area, Denning [5] presented that the data manipulated by a program can be typed with security levels, which naturally assumes the structure of a partially ordered set. However, he did not suggest how to automate the analysis. Millen [16] first built a formal correspondence between non-interference and mutual information, and established a connection between Shannon’s information theory and state-machine models of information flow in computer systems. McLean presented a very general Flow Model in [15]. The main weakness of this model is that it is too strong to distinguish between statistical correlation of values and casual relationships between high and low object. McIver and Morgan [14] devised a new information theoretic definition of information flow and channel capacity. Our method in this paper is more inspired by [4,13]: Clark, Hunt, and Malacaria [4] presented a reasonable quantitative analysis for a particular program in imperative languages but the bounds for loops are over pessimistic. McCamant and Ernst [13] investigated techniques for quantifying information flow revealed by complex programs by building flow graphs and considering the weight of the maximum flow over it but this is not a measurement in information theoretical aspect.

6 Conclusions

We have introduced a definition of quantitative program dependence graph, which can be used to capture the quantified dependencies of programs given the source node and target node of interest. We presented the semantics and an algorithm to build a chopped QPDG, and suggested a method to provide a further abstraction on the QPDG given the target node and source node by doing backward slicing. We argued that the reduced QPDG can be applied in the security community to quantify the information leakage from high level sensitive input to the low level public output.

References


