Measuring the Strength of Information Flows in Programs

WES MASRI
American University of Beirut
and
ANDY PODGURSKI
Case Western Reserve University

Dynamic information flow analysis (DIFA) was devised to enable the flow of information among variables in an executing program to be monitored and possibly regulated. It is related to techniques like dynamic slicing and dynamic impact analysis. To better understand the basis for DIFA, we conducted an empirical study in which we measured the strength of information flows identified by DIFA, using information theoretic and correlation-based methods. The results indicate that in most cases the occurrence of a chain of dynamic program dependences between two variables does not indicate a measurable information flow between them. We also explored the relationship between the strength of an information flow and the length of the corresponding dependence chain, and we obtained results indicating that no general relationship exists between the length of an information flow and its strength. Finally, we investigated whether data dependence and control dependence make equal or unequal contributions to flow strength. The results indicate that flows due to data dependences alone are stronger, on average, than flows due to control dependences alone. We present the details of our study and consider the implications of the results for applications of DIFA and related techniques.

Categories and Subject Descriptors: D.4.6 [Operating Systems]: Security and Protection - Information flow controls; D.2.4 [Software Engineering]: Software/Program Verification - Statistical Methods; D.2.5 [Software Engineering]: Testing and Debugging - Debugging aids, Monitors, Testing tools
General Terms: Security, Experimentation, Measurement
Additional Key Words and Phrases: Dynamic information flow analysis, information flow strength, information flow length, information leakage, program dependence, entropy, correlation, dynamic slicing

This research was supported in part by NSF grant CCR-0098325 (Dr. Podgurski) and LNCSR grant 022136 (Dr. Masri).
This paper is a revised, expanded version of a paper presented at the Fourth International Workshop on Dynamic Analysis (WODA 2006), Shanghai, China, May 2006.
Authors' addresses: W. Masri, Computer Science Department, American University of Beirut, Beirut, Lebanon 1107-2020, email: wm13@aub.edu.lb; A. Podgurski, Electrical Engineering and Computer Science Department, Case Western Reserve University, Cleveland, OH 44106, USA, email: andy@eecs.case.edu.
Permission to make digital/hard copy of part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date of appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2001 ACM 1073-0516/01/0300-0034 $5.00

1. INTRODUCTION

Informally, an information flow occurs from a variable x to variable y during execution of a program if observing the value of y at some point reduces one’s uncertainty about the value x took on at an earlier point, which indicates that y is probabilistically dependent on x. For example, sequential execution of the statements \( z = x/2; \ y = z - 1 \) will induce an information flow from x to y, because the final value of y is \( x/2 - 1 \), but the execution of z
Information flows between variables in a software system are relevant to the system’s security because they may indicate either leakage of sensitive information, i.e., confidentiality violations, or tampering with such information, i.e., integrity violations. For example, a flow of information from a database containing credit card account information to an untrusted socket connection may indicate an information leak. Information flows are also relevant in software testing, debugging, and maintenance because they indicate that certain program states, including erroneous ones, can affect certain subsequent states.

Returning to the topic of security, an information flow (security) policy for a software system specifies the variables between which it is permissible for information to flow in the system. An information flow policy must be enforced in order to preserve the confidentiality and integrity of data. Information flow analysis is a program analysis technique invented to support enforcement of information flow policies. It entails analyzing the information flows that can occur between program variables during execution [Denning and Denning 1977; Fenton 1974]. Both static [Denning 1976; Denning and Denning 1977; Sabelfeld and Myers 2003] and dynamic [Fenton 1974; Masri et al. 2004] forms of information flow analysis have been developed. The two forms are complementary. Dynamic information flow analysis is less prone to false positive indications of insecure flows than static information flow analysis, because conditional branches, pointer references, and array references are resolved at runtime. However, only static information flow analysis can be used to certify that an insecure flow reflected by static program dependences cannot possibly occur in a program. This paper addresses only dynamic information flow analysis.

Dynamic information flow analysis (DIFA) is based on the assumption that the occurrence of a chain of dynamic program dependences between two variables implies that information actually flows between them. That is, it is assumed that information actually flows from variable $x$ to variable $y$ whenever there is a sequence of $n \geq 2$ program actions (statement executions) $a_1, a_2, \ldots, a_n$ such that $x$ is used by $a_1$, $y$ is defined by $a_n$, and for $i = 1, 2, \ldots, n - 1$, $a_{i+1}$ is dynamically data dependent or dynamically control dependent on $a_i$. This assumption clearly holds in many instances. However, it is not obvious that it holds generally, because DIFA involves monitoring only the control flow and data flow in a running program; it does not consider the semantics of program statements or attempt to measure information flow. For example, when encrypting a

---

1 In this paper, the term variable refers to a local variable, a global variable, an array element, a class instance (i.e., object), or an instance variable.
message using the Advanced Encryption Standard (AES) [NIST 2001], many dependence chains must exist between the plaintext and ciphertext, but by simply observing the ciphertext it would be almost impossible to infer the plaintext; i.e., no discernable flow of information occurs from the plaintext to the ciphertext.

In addition to information flow policy enforcement, a number of other applications of dynamic dependence analysis in software engineering involve the assumption (often unstated) that the occurrence of a chain of dynamic program dependences linking program variables or statements indicates real information flows between them. For example, dynamic slicing [Agrawal and Horgan 1990; Tip 1995; Zhang et al. 2004; Masri et al. 2004] is based on the assumption that dynamic dependences indicate which program statements could have caused an observed program state to be erroneous. Dynamic impact analysis [Law and Rothermel 2003; Apiwattanapong et al. 2005] is based on the assumption that dynamic dependences indicate which parts of a program are affected by code changes made during maintenance. Dynamic taint analysis [Clause and Orso 2007; Newsome and Song 2005] is based on the assumption that dynamic dependences indicate which program operations could be affected by untrustworthy input values. Finally, testing techniques that employ dynamic information flow analysis [Masri et al. 2007] are based on the assumption that dynamic dependences reflect information flows that are relevant to the occurrence of program failures.

This paper describes an empirical study of the assumptions about dynamic dependences that underlie the aforementioned applications. The first question addressed by the study is: (Q1) Are dynamic program dependences generally indicative of measurable information flows in real programs? To try to answer this question, we did the following: (1) used DIFA to identify the apparent information flows that occurred during the execution of several programs, (2) computed the strength of the identified flows, i.e., measured the amount of information they propagated, and (3) analyzed the frequency distribution of the computed strengths.

The second aim of this work is to better understand the relationship between the strength of an information flow and its length, i.e., the number of direct data and/or control dependences in its dynamic dependence chain. Specifically, we have tried to answer the question: (Q2) Is the length of an information flow indicative of its strength? It seems plausible that the strength of information flows tends to attenuate as their length increases. If, for example, the average strength of information flows is inversely proportional to their length, then it may be advisable to give highest priority to the consideration of short flows. For example, in debugging a program one might examine
program actions in order of their proximity, in terms of dynamic dependences, to where an erroneous state was observed during execution. On the other hand, if long flows are frequently as strong as short ones then it is clearly risky to ignore long flows.

The third aim of this work is to answer the following question: (Q3) *Is there a difference between the strengths of information flows associated with data dependences and the strengths of information flows associated with control dependences?* This question is significant because it is possible that control dependences typically convey much less information than data dependences or vice versa.

In Masri and Podgurski [2006], we presented the results of a preliminary study that aimed at answering questions Q1 and Q2 using an entropy-based technique [Denning 1982] to compute the strengths of information flows. This study extends the previous work by primarily: (1) employing additional correlation-based techniques to compute the strengths of information flows, (2) using additional subject programs and larger data sets, and (3) examining whether there is a difference between the strengths of information flows associated with data dependences and the strengths of information flows associated with control dependences, i.e., answering Q3.

Section 2 describes how we used dynamic dependence analysis to identify information flows. Section 3 describes the entropy-based and correlation-based techniques we used to compute information flow strength. Section 4 describes our computation of information flow length. Section 5 presents our empirical results related to information flow strength and length. Section 6 discusses threats to the validity of our study results. Section 7 surveys related work. Finally, Section 8 presents our conclusions and future work.

2. DYNAMICALLY IDENTIFYING INFORMATION FLOWS

In the empirical study reported here we employed our dynamic information flow analysis tool [Masri et al. 2004], which is named *DynFlow*. It supports fine-grained DIFA by tracking flows involving local variables, global variables, array elements, class instances, and instance variables. For this study, we extended *DynFlow* to store additional information associated with each information flow, such as the values of its source and target variables and the length of its dependence chain. Next, we present the definitions and equations that underlie the algorithms employed by *DynFlow*; more formal and detailed versions are presented in Masri et al. [2004] and Masri [2004].
2.1 Dynamic Control Dependence
Informally, in the context of static analysis, a statement \( t \) is directly control dependent on a conditional statement \( s \), denoted \( t \ DCD \ s \), if \( s \) decides, via the branches it controls, whether \( t \) is executed or not [Ferrante et al. 1987; Podgurski 1989; Podgurski and Clarke 1990]. The dynamic counterpart of the \( DCD \) relation is the dynamic direct control dependence relation \( DDynCD \), which holds between program actions in an execution trace. The \( k^{th} \) action in a trace \( T \) is denoted by \( T(k) \) or by \( s^k \), where \( s \) is the corresponding statement [Korel 1994]. A subtrace of \( T \), denoted \( T(k, m) \) is the subsequence of actions starting at \( T(k) \) and ending at \( T(m) \). We now provide a formal definition of \( DDynCD \) that is based on the \( DCD \) relation:

**Definition.** Let \( s^k \) and \( t^m \) be two actions in an execution trace \( T \), where \( k < m \) and \( s^k \) is a predicate action. Then \( t^m \) is directly dynamically control dependent on \( s^k \), denoted \( t^m \ DDynCD \ s^k \), iff the subtrace \( T(k, m) \) demonstrates that \( t \ DCD \ s \). (Note that the dependent statement occurs in the trace after the statement it is dependent on; however, any information flow associated with the dependence is in the opposite direction.) The unique predicate action (if any) that \( t^m \) is directly dynamically control dependent on is denoted \( DDynCD(t^m) \).

Intuitively, \( DDynCD(t^m) \) is the most recent predicate action \( s^k \) to occur prior to action \( t^m \) such that \( t \ DCD \ s \). This is the key to precisely capturing dynamic control dependences because \( t^m \) may be in the dynamic scope of many predicate actions simultaneously, due to loops, nested control constructs, or unstructured flow of control.

2.2 Dynamic Data Dependence
Modeling dynamic data dependences between actions requires associating two sets of variables with each action: the set of variables \( D(s^k) \) defined (assigned a value) at \( s^k \), and the set of variables \( U(s^k) \) used (referenced) at \( s^k \).

**Definition.** Let \( s^k \) and \( t^m \) be two actions in an execution trace \( T \), where \( k < m \). Then \( t^m \) is directly dynamically data dependent on \( s^k \), denoted \( t^m \ DDynDD \ s^k \), iff

\[
(D(s^k) \cap U(t^m)) - D(T(k+1,m-1)) \neq \emptyset
\]

The set of actions that \( t^m \) is directly dynamically data dependent on is denoted \( DDynDD(t^m) \).

Informally, \( t^m \ DDynDD s^k \) iff \( t^m \) uses a variable that was last defined by \( s^k \). The \( DDynDD \) relation models both intra-procedural and inter-procedural data dependences.
The latter occur when an execution trace spans different functions and data defined in one function is used in another.

2.3 Direct Influence and Dynamic Information Flow Analysis

In addition to the $DDynCD$ and $DDynDD$ relations, we identify three related kinds of dynamic dependences between actions, each of which is inter-procedural (they are defined formally in [Masri 2004]):

1) Use of a value returned by a return statement
2) Use of a value passed via a formal parameter
3) Control dependence on a calling method’s invoke instruction (this is similar to the static entry-dependence effect described in [Sinha et al. 2000])

The combination of the aforementioned five types of data and control dependences comprises what we call “direct influence”. Given two actions $s^k$ and $t^m$ with $k < m$, $s^k$ directly influences $t^m$, denoted $s^k DInfluence t^m$, iff $t^m$ exhibits any of these five types of dependences upon $s^k$. The set of actions that $t^m$ is directly influenced by is denoted $DInfluence(t^m)$.

$DynFlow$’s dynamic information flow analysis algorithm is based on the following inductive equation, which characterizes the set of variables from which information is assumed to flow into an action $t^m$:

$$InfoFlow(t^m) = U(t^m) \cup \bigcup_{s^k \in DInfluence(t^m)} InfoFlow(s^k)$$

Here $U(t^m)$ is the set of variables used at $t^m$ and $DInfluence(t^m)$ is the set of actions that directly influence $t^m$. $InfoFlow(t^m)$ comprises the variables used at $t^m$ and all the variables used at actions that directly influence $t^m$.

In this paper, an information flow from variable $x$ defined at statement $s$ to variable $y$ defined at statement $t$ is uniquely identified by the 4-tuple $(x, s, y, t)$. Given that program execution might induce multiple instances of the same information flow, we identify each such instance by a 4-tuple $(x, s^k, y, t^m)$, where $s^k$ and $t^m$ are actions and $k < m$.

3. MEASURING THE STRENGTH OF INFORMATION FLOWS

In Section 2 we described how in our study we identified the information flows that occur during execution. This method is capable of identifying the information channel, i.e., the chain of dynamic data and control dependences, through which information can flow from one variable to another. In this section we are concerned with measuring the
amount of the information flowing through that channel, which we call information flow *strength*. We describe three techniques we used to measure information flow strength. The first technique is information theoretic and the other two are correlation-based. We opted to use multiple techniques in order to increase the evidence for our conclusions, although the information-theoretic technique is the most general one.

### 3.1 Entropy-based Technique

Our first technique for measuring the strength of information flows employs classical information theory [Cover and Joy 1991] to quantify the amount of information transferred between two variables. This technique was motivated by the entropy-based characterization of information flow presented in Denning [1982] and Bishop [2002].

The entropy of a random variable $X$, denoted $H(X)$, is a measure of uncertainty in $X$:

$$H(X) = -\sum_i P(X = x_i) \log_2 P(X = x_i)$$

(Note that by convention, $0 \times \log(0) = 0$, because $x \times \log(x) \to 0$ as $x \to 0$.)

The conditional entropy of $X$ given that $Y = y_j$, denoted $H(X \mid Y = y_j)$, is a measure of the remaining uncertainty about the value of $X$ when the value of the random variable $Y$ is known:

$$H(X \mid Y = y_j) = -\sum_i P(X = x_i \mid Y = y_j) \log_2 P(X = x_i \mid Y = y_j)$$

The expected value of this quantity with respect to the distribution of $Y$ is called the conditional entropy of $X$ given $Y$ and is denoted $H(X \mid Y)$:

$$H(X \mid Y) = -\sum_j P(Y = y_j) \left[ \sum_i P(X = x_i \mid Y = y_j) \log_2 P(X = x_i \mid Y = y_j) \right]$$

Let $c$ be a sequence of statement executions taking a program from state $\sigma$ to another state $\tau$ and let $x$ and $y$ be variables in the program. Suppose that $x$ and $y$ exist when the program is in state $\sigma$ and have the values $x_\sigma$ and $y_\sigma$, respectively. Suppose also that $y$ exists when the program is in state $\tau$ and has the value $y_\tau$. The sequence $c$ causes a *flow of information* from $x$ to $y$ if:

$$H(x_\sigma \mid y_\tau) < H(x_\sigma \mid y_\sigma)$$

That is, information flow occurs from $x$ to $y$ if one’s initial uncertainty about $x_\sigma$ based on knowing only $y_\sigma$ is reduced by executing $c$ and observing $y_\tau$. Note how this corresponds to the informal definition of information flow given in the Introduction.

---

*We assume here that a programs state consists of a program location and an assignment of values to all program variables.*
Following the analysis presented in Denning [1982] the amount of information transferred from $x$ (in state $\sigma$) to $y$ (in state $\tau$) is therefore:

$$\text{StrengthFlowEntropy}(x, y) = H(x_\sigma | y_\sigma) - H(x_\sigma | y_\tau)$$

Note that $\text{StrengthFlowEntropy}(x, y)$ will be greater than zero in the case of a measurable information flow between $x$ and $y$.

Clark et al. [2001; 2004] described a scenario where $\text{StrengthFlowEntropy}$ would yield a negative value and argued that the problem lies with the second term $H(x_\sigma | y_\tau)$ of the formula above, which accounts for the final observation of $y$ and assumes that the initial observation has been forgotten. They added that this is a safe assumption only if the observer has no memory, which is not generally the case. They provide their own modified version of $\text{StrengthFlowEntropy}$ that incorporates the observer’s memory:

$$\text{StrengthFlowEntropy}(x, y) = H(x_\sigma | y_\sigma) - H(x_\sigma | y_\tau, y_\sigma)$$

One of our main goals in this study is to evaluate whether the occurrence of a dynamic dependence chain between two variables implies that information actually flowed between them. That is, we want to quantify the amount of information that flowed from $x_\sigma$ to $y_\tau$ exclusively through channels involving dynamic dependence chains linking $x_\sigma$ to $y_\tau$. Any information acquired from outside of these channels must not be considered. Therefore $y_\sigma$ should have no role in our computation. This is equivalent to assuming either that $y$ does not exist in state $\sigma$ or that the observer has no means to access $y_\sigma$. For the purpose of our study we therefore arrive at the following measure of the strength of an information flow:

$$\text{StrengthFlowEntropy}(x, y) = H(x_\sigma) - H(x_\sigma | y_\tau)$$

$\text{StrengthFlowEntropy}(x, y)$ will be zero if no measurable flow occurred between $x$ and $y$, and it will be greater than zero otherwise. Note that by omitting $y_\sigma$, $\text{StrengthFlowEntropy}$, $\text{StrengthFlowEntropyD}$ and $\text{StrengthFlowEntropyC}$ become identical.

As an illustrative example, consider the two Java statements below:

\begin{verbatim}
  s1: a = x - x % 5;
  s2: y = a + 1;
\end{verbatim}

When $x$ takes on the values 3, 4 and 5, $y$ will be 1, 1, and 6, respectively. By repeatedly executing the above statements, the following $(x, y)$ value pairs might result: \{(3, 1), (3, 1), (3, 1), (4, 1), (4, 1), (4, 1), (5, 6)\}. Computing $\text{StrengthFlowEntropy}(x, y)$ yields a value of 0.5; suggesting a weak flow. This low value is expected since by
learning that \( y \) is 1 an observer is not absolutely certain of the value of \( x \), which could either be 3 or 4. As another example, consider the statements:

\[
\begin{align*}
& s_1: a = x - x \times 5; \\
& s_2: y = a + 1;
\end{align*}
\]

When \( x \) takes on the values 3, 4 and 5, \( y \) will be -11, -15, and -19, respectively, yielding the \((x, y)\) value pairs: \{(3, -11), (3, -11), (3, -11), (4, -15), (4, -15), (4, -15), (4, -15), (5, -19)\}. Computing \( \text{StrengthFlow}(x, y) \) based on these pairs yield a value of 1.4, suggesting a strong flow. Hence in this example, learning the value of \( y \) permits one to infer the value of \( x \) with more certainty.

Note how in the second example there is a clear one-to-one mapping between \( x \) values and \( y \) values, which is not the case in the first example. Intuitively, when the semantics of the code associated with a flow guarantee this type of one-to-one mapping the flow is likely to be strong. Identifying the program constructs and constraints that allow for such mapping is out of scope of this paper. In related work, Voas and Miller [1993] proposed the domain-to-range ratio (DRR) metric, which is the ratio of the cardinality of the possible inputs to the cardinality of the possible outputs. DRR provides an approximate measure of information loss between the input and output of a program or function. The authors hypothesized that an increase in DRR (higher information loss) leads to a decrease in testability and vice versa. This is consistent with our hypothesis that measuring the strength of information flows is relevant in software testing, debugging and related techniques.

### 3.2 Correlation-based Techniques

In our second technique for measuring the strength of an information flow we use the \textit{product moment} correlation coefficient (also known as \textit{Pearson’s} \( r \) or simply \textit{standard} \( r \)), which is typically used to measure the strength of the association between two linearly related variables [Kachigan 1986]. The \textit{standard} \( r \) coefficient, which ranges between -1 and +1, is defined as follows:

\[
r = \frac{\text{Cov}(x, y)}{\sigma_x \sigma_y}
\]

where

- \( \text{Cov}(x, y) \) = covariance of \( x \) and \( y \)
- \( \sigma_x \) = the standard deviation of \( x \)
- \( \sigma_y \) = the standard deviation of \( y \)

When the assumption of linearity is violated, \textit{standard} \( r \) will underestimate the strength of the relationship. Since this assumption is not likely to hold generally between
program variables, we will also employ another, possibly more appropriate, correlation coefficient. Our third technique for measuring the strength of an information flow uses the correlation ratio or eta coefficient [Siegel 1956; Krus 2006]. The eta coefficient yields the same value as standard r when the variables are linearly related and a greater value when they are nonlinearly related. That is, the difference between eta and standard r is a measure of the degree of nonlinearity of the relationship between two variables. The eta coefficient is defined below:

\[
eta = \frac{\sigma_{\bar{y}}}{\sigma_y}
\]

where

- \(\bar{y}\) = mean of the category that \(y\) belongs to (\(y\) values associated with the same \(x\) value belong to the same category)
- \(\sigma_{\bar{y}}\) = standard deviation of \(\bar{y}\)
- \(\sigma_y\) = standard deviation of \(y\)

Given a flow from \(x\) to \(y\), the strength measures computed using standard r and eta will be referred to by \(\text{StrengthFlow}_r(x, y)\) and \(\text{StrengthFlow}_{\text{eta}}(x, y)\), respectively. Note that \(\text{StrengthFlow}_r(x, y)\) is the absolute value of standard r and that in the case that either \(x\) or \(y\) is constant, we define \(\text{StrengthFlow}_r(x, y)\) to be zero, because there is no linear association between \(x\) and \(y\). Similarly, in the case that \(y\) is constant we define \(\text{StrengthFlow}_{\text{eta}}(x, y)\) to be zero, because there is no linear or nonlinear association between the variables.

As an illustration of the difference between \(\text{StrengthFlow}_r\) and \(\text{StrengthFlow}_{\text{eta}}\), consider the following (linearly related) pairs of \((x, y)\) values: \{(5, 100), (5, 100), (5, 100), (6, 200), (6, 200), (6, 200), (7, 300), (7, 300), (7, 300)\}. For these pairs, both \(\text{StrengthFlow}_r(x, y)\) and \(\text{StrengthFlow}_{\text{eta}}(x, y)\) are equal to 1.0, suggesting a strong flow. Now consider the following (nonlinearly related) pairs of values: \{(1, 5), (1, 5.5), (1, 5), (2, 0), (2, 0), (3, 6.5), (3, 6), (3, 6), (4, 1)\}. For these pairs, \(\text{StrengthFlow}_r(x, y)\) is equal to \(-0.11\), suggesting a weak flow, and \(\text{StrengthFlow}_{\text{eta}}(x, y)\) is equal to 0.99, suggesting a stronger flow. Finally, consider the following (non-related) pairs of values: \{(5, 1), (5, 2), (5, 3), (6, 1), (6, 2), (6, 3), (7, 1), (7, 2), (7, 3)\}. For these pairs, both \(\text{StrengthFlow}_r(x, y)\) and \(\text{StrengthFlow}_{\text{eta}}(x, y)\) are equal to 0.0, suggesting no flow of information between \(x\) and \(y\). Note though that \(\text{StrengthFlow}_{\text{eta}}\) is not necessarily always greater than \(\text{StrengthFlow}_r\). For the pairs \{(34, 0), (34, 0), (34, 10), (34, 10), (39, 0), (39, 32)\} the values of \(\text{StrengthFlow}_{\text{eta}}\) and \(\text{StrengthFlow}_r\) are 0.2 and 0.45, respectively. In this case
standard $r$ was more sensitive than $\eta$ in detecting the weak linear relationship between the two variables.

4. COMPUTING THE LENGTH OF INFORMATION FLOWS

The length of an information flow channel is the length of the chain of the dynamic data and control dependences it comprises. Let $o_{src}$ be the source variable and $o_{tg}$ the target variable of an information flow such that action $s^k$ last defined $o_{src}$ and action $t^m$ last defined $o_{tg}$. The associated flow instance is uniquely identified by the 4-tuple $(o_{src}, s^k, o_{tg}, t^m)$. Note that in the sequel the pairs $(o_{src}, o_{tg})$ and $(s^k, t^m)$ are used as shorthand for the 4-tuple $(o_{src}, s^k, o_{tg}, t^m)$.

We base our computation of the length of a flow channel on the following equation:

$$\text{LengthFlow} \ (s^k, t^m) = \begin{cases} 1, & o_{src} \in U (t^m) \\ \min (\text{LengthFlow} \ (s^k, r^n)) + 1, & o_{src} \notin U (t^m) \end{cases}$$

where $r^n \in \text{DInfluence} (t^m)$ and $o_{src} \in \text{InfoFlow} \ (r^n)$

The length is 1 if $i^m$ uses $o_{src}$; otherwise it is the length of the shortest dependence chain through which information from $o_{src}$ possibly flowed into $o_{tg}$.

Note that when our algorithm applies the above equation at an action $i^m$, all of the required sets and lengths have already been computed and are available. This makes it a forward computing algorithm and facilitates integrating it into DynFlow and executing it online. For example, given the snippet of Java code below:

```java
s1: a = x;
s2: b = a;
s3: y = a + b;
```

The following $\text{LengthFlow}$ computations will take place in tandem with the code execution:

At $s_1$:

$\text{LengthFlow}(x, a) = 1$

At $s_2$:

$\text{LengthFlow}(a, b) = 1$

$\text{LengthFlow}(x, b) = \min(\text{LengthFlow}(x, a)) + 1 = 2$

At $s_3$:

$\text{LengthFlow}(a, y) = 1$

$\text{LengthFlow}(b, y) = 1$

$\text{LengthFlow}(x, y) = \min(\text{LengthFlow}(x, a), \text{LengthFlow}(x, b)) + 1 = \min(1, 2) + 1 = 2$

Note how when computing $\text{LengthFlow}(x, y)$, $\text{LengthFlow}(x, a)$ and $\text{LengthFlow}(x, b)$ have already been computed and available for use.
5. EMPIRICAL STUDY

Our empirical study uses the techniques described in Sections 2, 3 and 4 to address the research questions mentioned in the Introduction:

Q1) Are dynamic program dependences generally indicative of measurable information flows in real programs?

Q2) Is the length of an information flow indicative of its strength?

Q3) Is there a difference between the strengths of information flows associated with data dependences and the strengths of information flows associated with control dependences?

Section 5.1 describes the subject programs used in the study. Section 5.2 discusses some of the design decisions we made. Sections 5.3, 5.4, and 5.5 present our empirical results concerning questions Q1, Q2, and Q3, respectively.

5.1 Subject Programs

We conducted our study using the XML parser Xerces 1.3, the XML pretty printer JTidy 3, the Servlet engine jigsaw 2.0.5, the Servlet/JSP engine Tomcat 3.0, the Servlet/JSP engine Tomcat 3.2.1 and the XML parser NanoXML 2 for Java. Each subject program was executed on numerous inputs of varied types and complexity. Xerces was executed using part of the XML Conformance Test Suite (www.w3.org/XML/Test), which is contributed by several organizations such as Sun and IBM. JTidy was executed using files downloaded from the Google Groups (groups.google.com) using a web crawler. jigsaw 2.0.5, Tomcat 3.0 and Tomcat 3.2.1, were executed using HTML, Servlet and JSP requests that were selected from (or based on) the examples included in the distribution of Tomcat 3.0. NanoXML was executed using files provided by the Software-artifact Infrastructure Repository (SIR) [Do et al. 2005].

Table I provides the following additional information about the subject programs: (1) number of lines of code, (2) size of test suites, (3) average execution-trace, which is approximated by the number of times the profiler interface of DynFlow was invoked, (4) average execution-time, which is mostly due to profiling and data collection/storage overhead, and (5) average size of the compressed execution profiles. As shown, most programs are sizable as they vary from 7,646 to 64,366 lines of code. The execution times and storage requirements ranged from 3 seconds and 54 KB for NanoXML to 683
seconds and 11,671 KB for JTidy. Also, the required computing resources varied markedly. For example, in order to collect and analyze the data for Xerces, Tomcat 3.0, Tomcat 3.2.1, and jigsaw, we used a 32-bit machine with 1 GB of RAM. For Xerces we used a 64-bit platform with 4 GB of RAM, and for JTidy we had to use a machine equipped with 16 GB of RAM.

### 5.2. Design Decisions

Computing any of the three strength measures described in Section 3 requires recording the values of the sources and targets of actual flows. In order to have a reasonable confidence in our computed measures, a minimal number of value pairs need to be used, i.e., for each flow, a minimal number of flow instances need to occur. Consequently, in our study we ignored flows that occurred fewer than 50 times in total, considering all executions. Table II shows for each subject program, the total number of identified flows and the number/percentage of flows used in our computations. Note how for Xerces and JTidy the majority of the flows had to be ignored and for the rest of the subject programs around half were ignored. Table II also shows, for each subject program, the maximum flow length observed amongst the used flows.

Table I. Statistics related to the subject programs.

<table>
<thead>
<tr>
<th>Subject Program</th>
<th># LOC</th>
<th># Tests</th>
<th>Exec-trace (avg.)</th>
<th>Exec-time (avg.)</th>
<th>Profile Size*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xerces</td>
<td>52,528</td>
<td>200</td>
<td>321,539</td>
<td>63 secs</td>
<td>2,533 KB</td>
</tr>
<tr>
<td>JTidy</td>
<td>9,153</td>
<td>50</td>
<td>2,732,189</td>
<td>683 secs</td>
<td>11,671 KB</td>
</tr>
<tr>
<td>Jigsaw</td>
<td>64,336</td>
<td>500</td>
<td>224,544</td>
<td>89 secs</td>
<td>1,065 KB</td>
</tr>
<tr>
<td>Tomcat 3.0</td>
<td>21,762</td>
<td>400</td>
<td>269,856</td>
<td>162 secs</td>
<td>868 KB</td>
</tr>
<tr>
<td>Tomcat 3.2.1</td>
<td>26,516</td>
<td>420</td>
<td>950,452</td>
<td>84 secs</td>
<td>1,380 KB</td>
</tr>
<tr>
<td>NanoXML</td>
<td>7,646</td>
<td>214</td>
<td>21,561</td>
<td>3 secs</td>
<td>54 KB</td>
</tr>
</tbody>
</table>

Table II. Statistics related to the observed flows (unrestricted flows) per subject program.

<table>
<thead>
<tr>
<th>Subject Program</th>
<th># Flows</th>
<th># Flows Used</th>
<th>% Flows Used</th>
<th>Max Flow Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xerces</td>
<td>1,926,325</td>
<td>175,909</td>
<td>9.13</td>
<td>254</td>
</tr>
<tr>
<td>JTidy</td>
<td>3,055,355</td>
<td>584,938</td>
<td>19.1</td>
<td>5,436</td>
</tr>
<tr>
<td>Jigsaw</td>
<td>200,365</td>
<td>115,719</td>
<td>57.7</td>
<td>201</td>
</tr>
<tr>
<td>Tomcat 3.0</td>
<td>115,452</td>
<td>68,545</td>
<td>59.37</td>
<td>463</td>
</tr>
<tr>
<td>Tomcat 3.2.1</td>
<td>252,089</td>
<td>106,400</td>
<td>42.2</td>
<td>872</td>
</tr>
<tr>
<td>NanoXML</td>
<td>15,779</td>
<td>7,333</td>
<td>46.5</td>
<td>139</td>
</tr>
</tbody>
</table>
Also, all three of our techniques require that the sources and targets of the studied flows be represented by scalar values. This is not a problem if the sources and targets are of primitive types, but in case of class instances, i.e., objects, we opted to do the following:

1) If the object is an instance of a class defined in the Java library, and the `java.lang.Object.hashCode()` method was overridden (e.g., as in `java.lang.String`), we assumed that the integer value returned by the `hashCode()` method represents the state of the object and therefore used it in our computations. This approach assumes that since the invoked hashing algorithm was designed and implemented by trusted library writers, it will rarely cause collisions. But given that collisions still might occur, there is a possibility that the results of our study are slightly inaccurate due to this decision.

2) If the object is an instance of a class defined in the Java library, but the `java.lang.Object.hashCode()` method was not overridden, or the object is an instance of a class that is not part of the Java library, we computed our own hash for that object to use with our techniques. The hash was computed by considering the states of the attributes of the studied object, and if need be, by recursively considering the attributes of its attributes and so on. The stopping condition was that either the attribute under consideration was of a primitive type or it fell into the category described in 1). Also in this case, given the possibility of collisions, the results of our study might slightly be inaccurate due to this design decision.

5.3. Q1: Are dynamic program dependences generally indicative of measurable information flows in real programs?

During the execution of a given subject program, for each detected flow instance we recorded the 6-tuple \((osrc, s, otgt, t, value(osrc, s^4), value(otgt, t^8))\) where \(osrc\) is the source identifier last defined at \(s^4\), \(otgt\) is the target identifier last defined at \(t^8\), \(value(osrc, s^4)\) is the source value generated by \(s^4\), and \(value(otgt, t^8)\) is the target value generated by \(t^8\). After all 6-tuples corresponding to all flow instances that occurred in all runs were saved, we consolidated them and computed estimates of the strength of each flow using \(\text{StrengthFlow}_{\text{Entropy}}(osrc, otgt)\), \(\text{StrengthFlow}_{\text{cdf}}(osrc, otgt)\) and \(\text{StrengthFlow}(osrc, otgt)\).

Figure 1 shows the histograms depicting the relative frequencies of estimated flow strengths for each combination of subject program and strength measure. Each chart shows interleaved histograms of one of the strength measures, where each histogram
corresponds to one of the subject programs. For example, the top chart shows the $StrengthFlow_{\text{Entropy}}$ histograms for each of the subject programs. The table at the bottom of Figure 1 shows, for each combination of subject program and strength measure, the average strength, maximum strength, and the percentage of flows having exactly zero strength\(^3\).

The following observations are made based on the charts of Figure 1:

1) Each of the $StrengthFlow_{\text{Entropy}}$ histograms exhibit sharply decreasing trends.

2) The $StrengthFlow_{\text{eta}}$ and $StrengthFlow_{\text{r}}$ histograms exhibit a sharp initial decrease, followed by a more gradual decrease, followed by an increase at high strength values. This might be due to the fact that the correlation based measures have an upper bound of 1.0 whereas the entropy based measure has no upper bound.

3) In each case a very large percentage of the flows had estimated strength close to or equal to zero.

4) The $StrengthFlow_{\text{eta}}$ histograms exhibit higher percentages of strong flows than the $StrengthFlow_{\text{r}}$ histograms do. This might be due to the fact that $\text{eta}$ is capable of detecting nonlinear relationships whereas $\text{standard r}$ is not.

We also make the following observations based on the table in Figure 1:

5) For the $StrengthFlow_{\text{Entropy}}$ measure, the percentages of flows with strength zero exhibited by Xerces, Jtidy, Tomcat 3.2.1, NanoXML, jigsaw, and Tomcat 3.0, are 63.76%, 82.25%, 91.21%, 92.21%, 94.61%, and 97.58%, respectively.

6) For the $StrengthFlow_{\text{eta}}$ measure, these percentages range from 63.68% to 97.48%, whereas for the $StrengthFlow_{\text{r}}$ measure they range from 63.8% to 97.59%.

7) For each program, the percentage of flows with estimated strength zero varied little between the three measures.

8) Except in the case of NanoXML, the $StrengthFlow_{\text{eta}}$ averages were higher than the $StrengthFlow_{\text{r}}$ averages. This might be due to the fact that $\text{eta}$ is capable of detecting nonlinear relationships whereas $\text{standard r}$ is not.

9) The pattern of variation in the strength averages between programs was similar for all three strength measures. The same can be said about the percentages of flows with estimated strength zero. This suggests that $StrengthFlow_{\text{Entropy}}$.

---

3 The left-most bar of the corresponding histogram is greater than or equal to this percentage as it represents the percentage of flows having a strength that fall in a range starting at zero, inclusive.
StrengthFlow\textsubscript{eta}, and StrengthFlow\textsubscript{r} are similarly effective for measuring the relative strengths of given flows.

10) Some flows exhibited very high strength measures, but they were not common.

It has been previously proven that a necessary condition for a change to the function computed by statement $s_1$ to affect the execution behavior of statement $s_2$ (in a schema-theoretic sense) is the presence of static program dependences connecting $s_1$ to $s_2$ [Podgurski 1989; Podgurski and Clarke 1990]. The results above concerning zero-strength flows suggest that even the presence of dynamic dependences connecting $s_1$ to $s_2$ is not a sufficient condition for the state generated by $s_1$ to affect the state generated by $s_2$ (in the information-theoretic sense). Moreover, they suggest that an apparent insecure information flow detected by DIFA is actually very likely to be harmless, in the sense of transmitting no measurable information.

Finally, our results suggest that StrengthFlow\textsubscript{Entropy}, which has no upper bound, permits flow strength to be measured with higher resolution than the other two measures, which exhibited apparent “clumping” of strength measurements near their upper bound.

5.4. Q2: Is the length of an information flow indicative of its strength?

For the purpose of answering the above question we conducted another study similar to the one described in the previous section, but for each flow instance we stored the 7-tuple $(o_{src}, s, o_{tgt}, t, value(o_{src}, s^t), value(o_{tgt}, t^o))$, where $l$ is the length of the flow instance computed using the definition of LengthFlow$(o_{src}, o_{tgt})$. Note that a flow defined by $(o_{src}, s, o_{tgt}, t)$ could be associated with multiple lengths, since in different flow instances, information could flow from a given source to a given target via different paths.

When all 7-tuples corresponding to all flow instances that occurred in all runs were saved, we consolidated them and computed estimates of StrengthFlow\textsubscript{Entropy}\left(o_{src}, o_{tgt}\right), StrengthFlow\textsubscript{eta}\left(o_{src}, o_{tgt}\right), and StrengthFlow\left(o_{src}, o_{tgt}\right) for each set of 50 or more instances sharing the same source, target, and length, i.e., the same $(o_{src}, s, o_{tgt}, t, l)$. Finally, we computed the average of the estimated strengths for each length $l$.

The frequency distributions of the flow lengths for all subject programs are shown in Figure 2. The lengths of most flows ranged between 2 and 50, but as indicated in Table II some flows were much longer, e.g., JTidy and Tomcat 3.2.1 had flows with lengths of 5,436 and 872, respectively. Note that the length is computed in terms of dependences involving byte code instructions rather than Java statements.

Figure 3 shows, for each combination of subject program and strength measure, the relationship between the lengths and the estimated strengths of the observed flows. There
is one chart for each of the three strength measures, which includes a plot of length vs. strength for each of the subject programs.

Examination of the charts does not reveal any consistent pattern in the relationship between length and average strength:

1) For NanoXML, jigsaw and Tomcat 3.0, very short flows exhibited higher average strengths than long flows.
2) For JTidy, very long flows exhibited higher average strengths than short flows.
3) For Xerces, long flows and flows of intermediate length were strongest.
4) For Tomcat 3.2.1, the average strength fluctuated as the length increased.
5) For Tomcat 3.0, jigsaw, and to some extent NanoXML, the average strength decreased as the length increased.

These results suggest that, in general, the length of an information flow is not indicative of its strength. This in turn suggests that long flows cannot be safely disregarded in applications of DIFA and related techniques like dynamic slicing.

Finally, the plots in the middle and bottom charts, especially those for Xerces and Tomcat 3.2.1, indicate that the values of \( \text{StrengthFlow}_{\text{eta}} \) tend to be higher than those of \( \text{StrengthFlow}_{\text{r}} \). This is expected in cases of nonlinear relationships.

5.5. Q3: Is there a difference between the strengths of information flows associated with data dependences and the strengths of information flows associated with control dependences?

To address question Q3, we took the following steps:

1) We executed our subject programs on their respective test suites with DynFlow configured to ignore the intra-procedural and inter-procedural control dependence components of DIInfluence, i.e., to ignore DDynCD and InterprocCD.
2) We executed our subject programs on their respective test suites with DynFlow configured to ignore data dependences involving computation-uses, i.e., to ignore DDynDD relationships involving variable uses within computational expressions (but not variable uses within conditional expressions).
3) We computed the strengths of the flows identified in 1) but not in 2). These flows, which are strictly due to data dependence, will be called DD-flows hereafter.
4) We computed the strengths of the flows identified in 2) but not in 1). These flows, which are strictly due to control dependence, will be called CD-flows hereafter.

As described in Section 2, an information flow is uniquely identified by the 4-tuple \((x, s, y, t)\) where \(x\) is the source defined at \(s\) and \(y\) is the target defined at \(t\). Some flows,
sharing the same unique identifier, were identified in both 1) and 2). We chose to ignore these flows, which we call dual flows, because they do not help to distinguish the effects of control dependences from those of data dependences. The flow \((x, s_1, y, s_3)\) induced by the execution of the following code fragment is an example of a dual flow:

\[
\begin{align*}
  s_1: & \quad x = 1; \\
  s_2: & \quad \text{if } (x > 0) \{ \\
  s_3: & \quad y = x;
\}
\end{align*}
\]

Notice that when the path \(s_1, s_2, s_3\) is executed, the flow \((x, s_1, y, s_3)\) is realized by (1) a direct dynamic data dependence involving \(s_1\) and \(s_3\) and (2) an indirect dependence composed of a direct dynamic data dependence involving \(s_1\) and \(s_2\) and a direct dynamic control dependence involving \(s_2\) and \(s_3\).

As before, in this experiment, we also ignored any flows that occurred fewer than 50 times. In order to facilitate our subsequent discussions, we use the term unrestricted flows to refer to the flows identified by DynFlow when it is configured to use all the components of DInfluence, i.e., flows due to data and/or control dependence as described in Section 2.3.

Table III shows for each subject program: 1) the total number of identified DD-flows, 2) the number/percentage of DD-flows we used to compute statistics, 3) the number of dual flows, and 4) the maximum DD-flow length observed among the flows we used to compute statistics. Table IV is similar to Table III except that it shows data associated with CD-flows. Examining Tables II, III, and IV, we observe the following:

1) Many dual flows were identified.
2) As expected, for each application the total number of unrestricted flows was larger than the numbers of DD-flows and CD-flows, respectively. Also, for each application the total number of unrestricted flows was larger than the numbers of DD-flows and CD-flows combined.
3) JTidy had DD-flows and CD-flows that were longer than any of its unrestricted flows. Similarly, jigsaw had CD-flows that were longer than any of its unrestricted flows.

The following code fragment illustrates how between a given pair of statements there may be an unrestricted flow that is shorter than any DD-flows. When the code executes, the length of the DD-flow \((x, s_1, y, s_4)\) is 2, whereas the length of the unrestricted flow \((x, s_1, y, s_4)\) is 1 (due to the control dependence induced by the branch):

\[
\begin{align*}
  s_1: & \quad x = 1; \\
  s_2: & \quad a = x; \\
  s_3: & \quad \text{if } (x > 0) \{
\end{align*}
\]
The next code fragment illustrates how between a given pair of statements there may be an unrestricted flow that is shorter than any CD-flows. When the code below executes, the length of the CD-flow \((x, s_1, y, s_4)\) is 2, but the length of the unrestricted flow \((x, s_1, y, s_4)\) is 1 (due to the data dependence induced by the assignment).

\[
\begin{align*}
s_1 &: x = 1; \\
s_2 &: \text{if } (x > 0) \{ \\
s_3 &: \text{if } (z > 0) \{ \\
s_4 &: y = x; \\
& \}
\end{align*}
\]

The primary concern of this section is to contrast the contributions of data dependences and control dependences to the strengths of information flows. Figure 4 shows, for each combination of subject program and strength measure, the percentages of strong unrestricted flows, strong DD-flows, and strong CD-flows. Using the entropy based strength measure (top chart), we consider a flow to be strong if \(\text{StrengthFlow}_{\text{Entropy}} > 1.0\). Using \(\text{eta}\) and \(\text{standard r}\), we consider a flow to be strong if \(\text{StrengthFlow}_{\text{eta}} > 0.5\) and \(\text{StrengthFlow}_{\text{r}} > 0.5\), respectively. In all cases, the percentages of strong DD-flows are considerably higher than the percentages of strong CD-flows. Also, except for \textit{Xerces}, the percentages of strong DD-flows are higher than the percentages of strong unrestricted flows.

Similarly, Figure 5 shows, for each combination of subject program and strength measure, the strength averages for unrestricted flows, DD-flows and CD-flows, respectively. In all cases, the averages for DD-flows are higher than the averages for CD-flows. Also, except for \textit{Xerces}, the averages for DD-flows are higher than the averages for unrestricted flows.

The results shown in Figures 4 and 5 clearly suggest that flows due to data dependences alone are stronger, on average, than flows due to control dependences alone.

Finally, the observations and conclusions we arrived at in Sections 5.3 and 5.4 regarding unrestricted flows seem to also hold for DD-flows and CD-flows. Figures 6 through 11 in Appendix A show supporting data that parallels the data presented in the previous two sections.
Fig. 1. Information flow strength frequency distributions for the subject programs using all three measures.
Fig. 2. Information flow length frequency distributions for the subject programs.

Fig. 3. Information flow length vs. average strength relationships for all subject programs using all three measures.
Table III. Statistics related to the observed DD-flows per subject program.

* Maximum length for DD-flows is larger than it is for unrestricted flows

<table>
<thead>
<tr>
<th>Program</th>
<th># Flows</th>
<th># Dual Flows</th>
<th>% Flows Used</th>
<th>Max Flow Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xerces</td>
<td>66,832</td>
<td>4,157</td>
<td>24,382</td>
<td>36.5</td>
</tr>
<tr>
<td>JTIdy</td>
<td>243,828</td>
<td>53,435</td>
<td>29,696</td>
<td>12.2</td>
</tr>
<tr>
<td>Jigsaw</td>
<td>23,926</td>
<td>1,642</td>
<td>13,368</td>
<td>55.8</td>
</tr>
<tr>
<td>Tomcat 3.0</td>
<td>27,666</td>
<td>3,261</td>
<td>9,585</td>
<td>34.6</td>
</tr>
<tr>
<td>Tomcat 3.2.1</td>
<td>24,474</td>
<td>2,483</td>
<td>13,111</td>
<td>53.6</td>
</tr>
<tr>
<td>NanoXML</td>
<td>4,420</td>
<td>411</td>
<td>1,476</td>
<td>33.4</td>
</tr>
</tbody>
</table>

Table IV. Statistics related to the observed CD-flows per subject program.

* Maximum length for CD-flows is larger than it is for unrestricted flows

<table>
<thead>
<tr>
<th>Program</th>
<th># Flows</th>
<th># Flows Used</th>
<th>% Flows Used</th>
<th>Max Flow Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xerces</td>
<td>90,609</td>
<td>11,107</td>
<td>12.26</td>
<td>90</td>
</tr>
<tr>
<td>JTIdy</td>
<td>957,945</td>
<td>154,615</td>
<td>16.1</td>
<td>6,121*</td>
</tr>
<tr>
<td>Jigsaw</td>
<td>17,988</td>
<td>11,917</td>
<td>66.2</td>
<td>333*</td>
</tr>
<tr>
<td>Tomcat 3.0</td>
<td>12,488</td>
<td>9,166</td>
<td>73.4</td>
<td>463</td>
</tr>
<tr>
<td>Tomcat 3.2.1</td>
<td>48,698</td>
<td>20,639</td>
<td>42.4</td>
<td>256</td>
</tr>
<tr>
<td>NanoXML</td>
<td>2,045</td>
<td>1,742</td>
<td>85.2</td>
<td>50</td>
</tr>
</tbody>
</table>
Fig. 4. Percentages of strong unrestricted flows, DD-flows and CD-flows.
Fig. 5. Strength averages of unrestricted flows, DD-flows and CD-flows.
6. THREATS TO VALIDITY

Because our study employed just a few subject programs, it is not possible to draw broad conclusions based on the results, despite the sizable data sets. To address this threat to external validity, further empirical studies with a variety of other subject programs and data sets are clearly needed. Ideally, the subject programs would constitute a representative sample of real systems, applications, utilities, and components. However, since no investigators have the ability to access and utilize any software they wish, it will be necessary to settle for a sample of software that is available and practical for researchers to employ.

The internal threats to the validity of our study stem from some of the design decisions we made (see Section 5.2):

1) The fact that we ignored flows that occurred fewer than 50 times lead us, in some cases, to discard a considerable percentage of the observed flows (see Table II). We investigated the impact of this design decision by conducting our experiments for a second time while ignoring only flows that occurred fewer than 10 times. The results obtained were similar to the original results and would not lead us to change our conclusions. Table V compares, for each subject program, the percentages of flows used in the case when the cut-off is set to 10 and 50, respectively. As shown, ignoring only flows that occurred fewer than 10 times caused a considerable relative increase in the number of used flows in most cases.

2) In order to reduce the number of ignored flows, we collected value pairs from within and across executions. One might instead base the strength computations on each execution by itself and then average the results across executions, but this would lead to many flows being ignored.

Table V. Comparison of the numbers of flows used in the case when the cut-off is set to 10 and 50, respectively.

<table>
<thead>
<tr>
<th>Program</th>
<th># Flows</th>
<th># Flows Used (cut-off = 10)</th>
<th>% Flows Used (cut-off = 10)</th>
<th>% Flows Used (cut-off = 50)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xerces</td>
<td>1,926,325</td>
<td>461,138</td>
<td>23.9</td>
<td>9.13</td>
</tr>
<tr>
<td>JTidy</td>
<td>3,055,355</td>
<td>885,671</td>
<td>28.9</td>
<td>19.1</td>
</tr>
<tr>
<td>Jigsaw</td>
<td>200,365</td>
<td>152,922</td>
<td>76.3</td>
<td>57.7</td>
</tr>
<tr>
<td>Tomcat 3.0</td>
<td>115,452</td>
<td>77,582</td>
<td>67.2</td>
<td>59.37</td>
</tr>
<tr>
<td>Tomcat 3.2.1</td>
<td>252,089</td>
<td>158,860</td>
<td>63.0</td>
<td>42.2</td>
</tr>
<tr>
<td>NanoXML</td>
<td>15,779</td>
<td>9,686</td>
<td>61.4</td>
<td>46.5</td>
</tr>
</tbody>
</table>
3) When an object is involved in a flow, our technique uses/computes the object’s hash and uses it to represent its state. The possibility of collisions might make the results of our study slightly inaccurate. Note that the task of identifying object states is not trivial [Dallmeier et al. 2006; Xie et al. 2006; Visser et al. 2003].

7. RELATED WORK

Lampson [1973] motivated research on information flow analysis by describing the problem and listing a number of possible information leaks. Fenton [1974] proposed an abstract machine called the Data Mark Machine to support dynamic checking of information flows. Denning and Denning [1976; 1977] proposed a technique based on static control flow and data flow analysis for verifying a program’s compliance with an information flow policy. A number of language-based static type checking systems have also been proposed [Sabelfeld and Myers 2003]. In such systems every program expression is assigned a security type in addition to its ordinary type. In type checking a program, the compiler ensures that the program cannot exhibit insecure information flows at run-time.

In Masri et al. [2004] we presented a new forward-computing approach to DIFA that can be used to detect, prevent, and debug insecure information flows in programs. (Forward computing algorithms operate in tandem with program execution and have the advantage that they do not require a previously stored execution trace.) A prototype tool implementing the approach for Java byte code programs was also described. This tool, which we call DynFlow, can be used offline with either synthetic test cases or captured operational inputs [Steven et al. 2000] to detect violations of information flow policies, and it can also be used online with certain applications. Online DIFA is possible only with a forward computing algorithm. In Masri et al. [2007] we used DynFlow to generate information flow and slice profiles that were used in test case filtering experiments. In Masri and Podgurski [2005] we used it to demonstrate the utility of DIFA for online signature-based and offline anomaly-based intrusion detection.

Clark et al. [2001] used information theory to provide a formal approach to analyze the amount of confidential information which may be leaked by programs. Although their approach can be automated it applies only to programs written in a very simple imperative language that contains no iterations. Lowe [2002] quantified the amount of information passed through covert (timing) channels. His proposed approach is based

---

4 DynFlow can be used online with applications that are not processing-intensive and do not have demanding performance requirements.
upon counting the number of different behaviors of a high level user that can be
distinguished by a low level user. Note that our study did not involve information flows
through covert channels and our tool is not capable of detecting them.

In Binkley and Harman [2004] the dependence of predicates on global variables and
formal parameters was empirically studied. The results show that as the number of
formal parameters available to a predicate increases, there is a decrease in the proportion
of these formal parameters that influence the predicate. No such correlation was found
for global variables. Evidence is also presented that the globals and formals are
independent of one another and that their numbers are independent of the size of the
procedure that contains them.

In Voas and Miller [1993] the domain-to-range ratio (DRR) was proposed, which
provides an approximate measure of information loss between the input and output of a
program or function. It is the ratio of the cardinality of the possible inputs to the
cardinality of the possible outputs. The authors hypothesized that an increase in DRR
(higher information loss) leads to a decrease in testability and vice versa. Naturally, this
is consistent with our hypothesis that the strength of information flows is relevant to
software testing and debugging. Later, Woodward and Al-Khanjari [2000] established a
mathematical link between DRR and program testability.

More recently, McCamant and Ernst [2006] presented a tainting technique for
dynamically tracking information flow in C programs. Their technique quantitatively
measures information flow at the instruction level using bit-tracking analysis. In
subsequent work [2007] they proved the soundness of their proposed dynamic
information analysis through a simulation-based proof technique. Also, Clause et al.
[2007] presented Dytan, a tool for dynamic tainting analysis that supports x86
executables.

Zhang et al. [2006] demonstrated that by switching a predicate’s outcome (i.e.,
altering the control flow) it is often possible to change the outcome of an execution from
failing to passing. They presented evidence that in the union of the backward and
forward dynamics slices of the switched predicate the faulty statement is often located
close to the predicate in terms of dependence distance. In Zhang et al. [2005] they
presented evidence that an effective strategy for using dynamic slices in debugging is to
examine the statements in a backward dynamic slice in increasing order of their
dependence distance from the observed incorrect output. Zhang et al’s results concerning
the significance of dependence distance appear to be at odds with our results concerning
the relationship between the lengths and strengths of information flows. However, their
results may be due to very short flows with high strengths, which we observed with some of our subject programs. Further study will be necessary to resolve this question. Finally, Zhang et al. [2006] present a heuristic technique for reducing the size of a dynamic slice, which is based on consideration of the values that variables in the slice may have. This technique can be viewed as seeking to identify statements from which there are strong information flows to incorrect outputs but not to correct outputs.

8. CONCLUSION AND FUTURE WORK

In conclusion, the results of our empirical study suggest the following:

1) The existence of dynamic program dependence between two variables does not indicate that with high probability there is a measurable information flow between them. That is, dynamic program dependence is not necessarily indicative of actual information flow in real programs.

2) No general relationship exists between the length of an information flow and its strength. That is, the length of an information flow is not indicative of its strength.

3) Flows due to data dependences alone are stronger, on average, than flows due to control dependences alone.

4) Apparent information flows identified by analysis of dynamic dependences have a high probability of being spurious, i.e., being false positives.

5) Long flows are not generally less significant than short flows and should not be dismissed without further examination.

Results (1) and (4) suggest that dynamic information flow analysis and its close relatives taint analysis, dynamic slicing, and dynamic impact analysis are actually quite imprecise, although they are still more precise in some respects than their static counterparts. Estimating information flow strength as we have done makes it possible to take a calculated risk and ignore apparently zero-strength flows in applications such as intrusion detection, debugging, dynamic impact analysis, and test-case filtering in order to focus on bona-fide flows. Alternatively, empirical flow strengths could be monitored for anomalous changes which might indicate intrusions or changes in program usage.

Results (2) and (5) call into question the idea of giving priority to short flows in applications of DIFA and related techniques. If these results are confirmed, it will, unfortunately, invalidate an attractive heuristic for reducing analysis costs and manual effort. We caution, however, that definitively confirming our results and understanding their full implications for specific applications will require substantial further study.
Although our results may be surprising, they become less so when one considers that dynamic dependence analysis does not take into account the semantics of program statements or the probability distribution of program inputs, which are critical to whether or not measurable information flows occur between particular variables. Some readers may wonder if the information-theoretic characterization of information flow in programs, which has been used for some time in software security research [Denning 1982; Bishop 2002], is appropriate to software engineering applications such as debugging, where causality (e.g., of program failures) is the primary concern, not statistical correlation. Although a comprehensive answer to this question will require further research, we note that statistically based techniques for discovering causal models from observational data have been developed by the machine learning community [Pearl 2000; Spirtes et al. 2000].

In addition to seeking to confirm the results presented here, in our future work we intend to investigate the following questions:

a) What is the most appropriate way to measure the strength of information flows for particular applications? It may desirable to define new strength measures for some applications, which can be interpreted more directly.

b) What are the characteristics of executions under which strong (or weak) flows occur? That is, can one determine whether a flow is likely to be strong or weak based on the way a program behaves when it occurs?

c) Is the significance of a flow really proportional to its strength? That is, should one focus on strong flows in applications of DIFA?

d) How is the strength/length ratio affected by factors such as program structure, coupling, cohesion, conformance to the “Law of Demeter” [Lieberherr 2004], and the presence or absence of defects?

Finally, we hope to leverage our conclusions about the strengths of information flows to improve existing techniques for taint analysis, information leak detection, dynamic slicing, test suite prioritization, impact analysis, and possibly others.

ACKNOWLEDGMENTS

We thank John Haddad and Ibrahim Abou-Faycal for their helpful comments and discussions.

REFERENCES


KRUS, D.J. 2006. Visual Statistics: Chapter 14, Regression on Multiple Categories, visualstatistics.net


The following figures show strength and length data associated with DD-flows and CD-flows. Note the similarity with the figures presented in Sections 5.3 and 5.4.
<table>
<thead>
<tr>
<th></th>
<th>Entropy</th>
<th></th>
<th></th>
<th>ETA</th>
<th></th>
<th></th>
<th>Standard r</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg.</td>
<td>Max</td>
<td>% Flows at zero</td>
<td>Avg.</td>
<td>Max</td>
<td>% Flows at zero</td>
<td>Avg.</td>
<td>Max</td>
</tr>
<tr>
<td>Xerces</td>
<td>0.18</td>
<td>9.96</td>
<td>91.44</td>
<td>0.056</td>
<td>1.0</td>
<td>91.43</td>
<td>0.05</td>
<td>1.0</td>
</tr>
<tr>
<td>JTidy</td>
<td>1.18</td>
<td>10.35</td>
<td>63.15</td>
<td>0.29</td>
<td>1.0</td>
<td>63.15</td>
<td>0.21</td>
<td>1.0</td>
</tr>
<tr>
<td>Jigsaw</td>
<td>0.163</td>
<td>8.78</td>
<td>94.47</td>
<td>0.046</td>
<td>1.0</td>
<td>94.50</td>
<td>0.04</td>
<td>1.0</td>
</tr>
<tr>
<td>Tomcat 3.0</td>
<td>0.35</td>
<td>8.74</td>
<td>90.29</td>
<td>0.075</td>
<td>1.0</td>
<td>90.26</td>
<td>0.035</td>
<td>1.0</td>
</tr>
<tr>
<td>Tomcat 3.2.1</td>
<td>0.29</td>
<td>8.74</td>
<td>84.85</td>
<td>0.12</td>
<td>1.0</td>
<td>84.86</td>
<td>0.07</td>
<td>1.0</td>
</tr>
<tr>
<td>NanoXML</td>
<td>0.12</td>
<td>6.42</td>
<td>93.97</td>
<td>0.054</td>
<td>1.0</td>
<td>93.97</td>
<td>0.052</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Fig. 6. DD-flows *strength* frequency distributions for the subject programs using all three measures.
Fig. 7. DD-flows length frequency distributions for the subject programs.

Fig. 8. DD-flows length vs. average strength relationships for all subject programs using all three measures.
Fig. 9. CD-flows strength frequency distributions for the subject programs using all three measures.
Fig. 10. CD-flows length frequency distributions for the subject programs.

Fig. 11. CD-flows length vs. average strength relationships for all subject programs using all three measures.