View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Queen Mary Research Online

Quantified Interference for a While Language

Clarke, Dave; Hunt, Sebastian; Malacaria, Pasquale

For additional information about this publication click this link.
http://qmro.qmul.ac.uk/jspui/handle/123456789/5036

Information about this research object was correct at the time of download; we occasionally make corrections to records, please therefore check the published record when citing. For more information contact scholarlycommunications@qmul.ac.uk

QUEEN MARY
AND WESTFIELD COLLEGE
UNIVERSITY OF LONDON

Department of Computer Science

Research Report No. RR-03-04 ISSN 1470-5559 October 2003

Quantified Interference for a While Language

David Clarke, Sebastian Hunt and Pasquale Malicaria

Quantified Interference for a While Language

David Clark
Dept of Computer Science
Kings College, London
david@dcs.kcl.ac.uk

Sebastian Hunt
Dept of Computing
City University, London
seb@soi.city.ac.uk

Pasquale Malacaria
Dept of Computer Science
Queen Mary, London
pm@dcs.qmul.ac.uk

ABSTRACT

We show how an information theoretic approach can quantify interference in a simple imperative language that includes a looping construct. In this paper we focus on a particular case of this definition of interference: leakage of information from private variables to public ones via a Trojan Horse attack. We show how in our setting Shannon's Perfect Secrecy theorem provides sufficient conditions to determine leakage of a program. The major result of the paper is a quantitative analysis for this language that employs a use-definition graph to calculate bounds on the leakage into each variable. Some improvements to these bounds are developed via improved results for some classes of expressions of the language.

1. INTRODUCTION

Mathematical quantitative tools (like probability theory and statistics) have played an increasing role both in the theory and practise of most sciences. However the theory (and theory based analysis) of software systems largely relies on logics and makes little use of quantitative mathematics.

Traditionally logic is a qualitative discipline, things are true or false, provable or not, typable or not. It is our belief however that some fundamental notions in theoretical computer science might benefit from a quantitative study.

Take the notion of *interference* [4, 10] between program variables, informally the capability of variables to affect the value of other variables. Absence of interference (non-interference) is often used in proving that a system is well-behaving, whereas interference can lead to mysterious (mis-)behaviours. However the misbehaviour in presence of interference will generally happen only when there is *enough* interference. Think in terms of electric current: non-interference between variables X, Y is the absence of a circuit involving X, Y; interference is the existence of a circuit; this however doesn't imply that there is enough "current" in the circuit to affect the behaviour of the system.

Concrete examples of this are provided by *access control* based software systems. To enter such a system the user has to pass an identification stage; whatever the outcome of this stage (authorisation or failure) some information has been leaked (in the case of failure the search space for the right key has now become smaller). Hence these systems present interference [4] so they are not "secure" in a qualitative sense. However, common sense suggests to consider them secure if the interference is very small.

The aim of this paper is to use Shannon's information theory [13] to define a quantified notion of interference for a simple imperative language and to derive a program analysis based on this notion.

The theoretical part of the paper investigates which of Shannon's measures (entropy, conditional entropy, mutual information) is the right one for the task and proves results relating interference of program variables, independence of random variables and leakage of confidential information.

We then use the theory as the basis for a program analysis which can be used to derive bounds on the quantity of confidential information leaked by a program.

In a previous paper [1] we sketched an information theory based program analysis for a simple language without loops. The achievements presented in this paper are:

theory: Formal relationships between non-interference, random variable independence and leakage of confidential data are established. Shannon's Perfect Secrecy theorem is revisited in light of these correspondences and interpreted as providing sufficient conditions for a program to be "secure".

analysis: The analysis is now graph and not syntax based. This allows us to handle loops (although crudely).

expressions: Improved bounds on general equality tests and arithmetic expressions are presented.

1.1 Related work

The work we describe in this paper is not the first attempt to apply information theory to the analysis of confidentiality properties. The earliest example of which we are aware is in Denning's book [3] where she gives some examples of how information theory may be used to calculate the leakage of confidential data via some imperative language program constructs. However she does not develop a systematic, formal approach to the question as we do in this paper. Another early example is that of Jonathan Millen [7] which points to the relevance of Shannon's use of finite state systems in the analysis of channel capacity. More recent is the work of James W. Gray [15], which develops a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

quite sophisticated operational model of computation and relates non-interference properties to information theoretic properties. However, neither of these deals with the analysis of programming language syntax, as we do here. Contemporary with our own work has been that of Di Pierro, Hankin and Wiklicky. Their interest has been to measure interference in the context of a probabilistic concurrent constraint setting where the interference comes via probabilistic operators. In [8] they derive a quantitative measure of the similarity between agents written in a probabilistic concurrent constraint language. This can be interpreted as a measure of how difficult a spy (agent) would find it to distinguish between the two agents using probabilistic covert channels, with a measure of 0 meaning the two agents were indistinguishable. Their approach does not deal with information in an information theoretic sense although the implicit assumption in example 4 in that paper is that the probability distribution of the value space is uniform.

By contrast, much more has been done with regard to syntax directed analysis of non-interference properties. See particularly the work of Sands and Sabelfeld [11, 12]. However we aren't aware of any graph based analysis using information theory.

2. INFORMATION THEORY AND INTERFERENCE

2.1 The language and its semantics

In a semantics-based analysis of security properties, there is a trade-off between tractability and accuracy. Any semantic model is, necessarily, an abstraction of its physical implementations and the limits of an analysis based on that model are determined by the nature of the abstraction. Put more concretely, a system which can be shown to be secure with respect to a semantic model may still be vulnerable to attacks which exploit precisely those aspects of its behaviour which are not modelled.

In this paper we consider a programming language with a simple operational semantics and we analyse confidentiality properties based purely on the input-output behaviour which this semantics defines. The guarantees provided by our analysis are correspondingly limited. In particular, our analysis addresses the question of how much an attacker may learn (about confidential information) by observing the input-output behaviour of a program, but does not tell us anything about how much can be learned from its running-time. We also neglect to deal with any complications which may arise due to termination behaviour, by assuming that all programs terminate on all inputs.

The language contains just the following control constructs: assignment, `while`-statements, `if`-statements, sequential composition. The left hand sides of assignments are variable identifiers, the right hand sides are integer or boolean expressions; `while` loops and `if`-statements involve boolean expressions in the standard way. We do not fully specify the language of expressions but we make the assumption that all expressions define total functions on stores. The language is deterministic and so, for each program P, the semantics induces (assuming termination) a total function $[\![P]\!] : \Sigma \to \Sigma$, where Σ is the domain of stores. A store $\sigma \in \Sigma$ is just a finite map from variable names to k-bit integers (integers n in the range $-2^{k-1} \leq n < 2^{k-1}$) and booleans.

Given a program P, a *program point* is either the special node ω (the *exit point*), or any occurrence in P of an assignment statement, `if`-statement or `while`-statement. We call the top-most program point ι (the *entry point*). The operational semantics is standard and defines a transition relation $\to$ on configurations (n, σ), where n is a program point and σ is a store. A *trace* is a sequence of configurations $(n_1, \sigma_1) \cdots (n_j, \sigma_j)$ such that $(n_i, \sigma_i) \to (n_{i+1}, \sigma_{i+1})$ for $1 \leq i < j$.

2.2 Degrees of interference

We suppose that the variables of a program are partitioned into two sets, H (*high*) and L (*low*). High variables may contain confidential information when the program is run, but these variables cannot be examined by an attacker at any point before, during or after the program's execution. Low variables do not contain confidential information before the program is run and can be freely examined by an attacker before and after (but not during) the program's execution. This raises the question of what an attacker may be able to learn about the confidential inputs by examining the low variable outputs.

One approach to confidentiality, quite extensively studied [4], is based on the notion of *non-interference*. This approach looks for conditions under which the values of the high variables have no effect on (do not 'interfere' with) the values of the low variables when the program is run. We can formalise non-interference in the current setting as follows. A program P is non-interfering if, whenever $\sigma_1\ X = \sigma_2\ X$ for all X in L, then $[\![P]\!]\sigma_1 = \sigma_1'$ and $[\![P]\!]\sigma_2 = \sigma_2'$ with $\sigma_1'\ X = \sigma_2'\ X$ for all X in L. If this condition holds, an attacker learns nothing about the confidential inputs by examining the low outputs.

Thus non-interference addresses the question of *whether or not* a program leaks confidential information. In the current work, by contrast, we address the question of *how much* information may be leaked by a program.

To help explore the difference between the approaches, consider the following two programs:

1. `if (H == X) Y = 0 else Y = 1 fi`
2. `if (H < X) Y = 0 else Y = 1 fi`

Here we specify that `H` is high while `X` and `Y` are low. Clearly, neither of these programs has the non-interference property, since the final value of `Y` is affected by the initial value of `h`. But are the programs equally effective from an attacker's point of view? Suppose we allow the attacker not only to examine but actually to *choose* the initial value of `X`. Suppose further that the attacker can run the program many times for a given choice of value for `h`. There are 2^k possible values which `h` may have and the attacker wishes to know which one it is. It is easy to see (below) that the second program is more effective than the first, but the significance of this difference depends on the *distribution* of the values taken by `h`.

At one extreme, all 2^k values are equally likely. Using the first program it will take the attacker, on average, 2^{k-1} runs, trying successive values for `X`, to learn the value of `h`. Using the second program, the attacker can choose values of `X` to effect a binary search, learning the value of `h` in at most k runs.

At the other extreme, `h` may in fact only ever take a few of the possible values. If the attacker knows what these few

values are, then both programs can clearly be used to find the actual value quickly, since the search space is small.

2.3 Information and conditional information

We use Shannon's information theory to quantify the amount of information a program may leak and the way in which this depends on the distribution of inputs. Shannon's measures are based on a logarithmic measure of the unexpectedness, or surprise, inherent in a probabilistic event. An event which occurs with some non-zero probability p is regarded as having a 'surprisal value' of:

$$\log \frac{1}{p}$$

Intuitively, surprise is inversely proportional to likelihood. The base for log may be chosen freely but it is conventional to use base 2 (the rationale for using a logarithmic measure is given in [13]). The total information carried by a set of n events is then taken as the weighted sum of their surprisal values:

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

(if $p_i = 0$ then $p_i \log \frac{1}{p_i}$ is defined to be 0). This quantity is variously known as the *self-information* or *entropy* of the set of events.

The events of interest for us are observations of the values of variables before and after the execution of (part of) a program. Suppose that the inputs to a program take a range of values according to some probability distribution. In this case we may use a random variable to describe the values taken (initially) by a program variable, or set of program variables. In sect 2.5 we show how to extend this idea, defining, for each program point, the random variable corresponding to a program variable at that point.

For our purposes, a random variable is a total function $X : D \to R$, where D and R are finite sets and D comes with a probability distribution (D is the sample space). We adopt the following conventions for random variables:

1. if X is a random variable we let x range over the set of values which X may take; if necessary, we denote this set explicitly by $R(X)$; the domain of X is denoted $D(X)$
2. we write $p(x)$ to mean the probability that X takes the value x; where any confusion might otherwise arise, we write this more verbosely as $P(X = x)$
3. for a vector of (possibly dependent) random variables $(X_1, \ldots, X_n)$, we write $p(x_1, \ldots, x_n)$ for the joint probability that the X_i simultaneously take the values x_i; equivalently, we may view the vector as a single random variable X with range $R(X_1) \times \cdots \times R(X_n)$
4. when summing over the range of a random variable, we write $\sum_x f(x)$ to mean $\sum_{x \in R(X)} f(x)$; again, we use the more verbose form where necessary to avoid confusion

The entropy of a random variables X is denoted $\mathcal{H}(X)$ and is defined, in accordance with (1), as:

$$\mathcal{H}(X) = \sum_x p(x) \log \frac{1}{p(x)} \tag{2}$$

Because of possible dependencies between random variables, knowledge of one may change the surprise (hence information) associated with another. This is of fundamental importance in information theory and gives rise to the notion of *conditional* entropy. Suppose that $Y = y$ has been observed. This induces a new random variable $X \upharpoonright (Y = y)$ (X restricted to those outcomes such that $Y = y$) with the same range as X but with domain $\{d \in D(X) : Y(d) = y\}$ and $P((X \upharpoonright (Y = y)) = x) = P(X = x | Y = y)$, where

$$P(X = x | Y = y) = \frac{p(x, y)}{p(y)}$$

The conditional entropy of X given knowledge of Y is then defined as the expected value (ie, weighted average) of the entropy of all the conditioned versions of X:

$$\mathcal{H}(X|Y) = \sum_y p(y) \mathcal{H}(X \upharpoonright (Y = y)) \tag{3}$$

A key property of conditional information is that $\mathcal{H}(X|Y) \leq \mathcal{H}(X)$, with equality iff X and Y are independent.

In particular, we are interested in how much of the information carried by the high inputs to a program can be learned by observation of the low outputs, assuming that the low inputs are known. Since our language is deterministic, any variation in the outputs is a result of variation in the inputs. Once we account for knowledge of the program's low inputs, therefore, the only possible source of surprise in an output is interference from the high inputs. Given a program variable (or set of program variables) X, let X^ι and X^ω be, respectively, the corresponding random variables on entry to and exit from the program. We take as a measure of the amount of leakage into X due to the program:

$$\mathcal{L}(X) = \mathcal{H}(X^\omega | L^\iota) \tag{4}$$

(recall that L is the set of low variables, thus L^ι is the random variable describing the distribution of the program's non-confidential inputs).

2.4 Mutual information

Information theory provides a more general way of measuring the extent to which information may be shared between two sets of observations. Given two random variables X and Y, the mutual information between X and Y, written $\mathcal{I}(X;Y)$ is defined as follows:

$$\mathcal{I}(X;Y) = \sum_x \sum_y p(x,y) \log \frac{p(x,y)}{p(x)p(y)} \tag{5}$$

Routine manipulation of sums and logs yields three equivalent ways of defining this quantity:

$$\mathcal{I}(X;Y) = \mathcal{H}(X) + \mathcal{H}(Y) - \mathcal{H}(X,Y) \tag{6}$$

$$\mathcal{I}(X;Y) = \mathcal{H}(X) - \mathcal{H}(X|Y) \tag{7}$$

$$\mathcal{I}(X;Y) = \mathcal{H}(Y) - \mathcal{H}(Y|X) \tag{8}$$

As shown by (6), $\mathcal{I}(X;Y)$ is symmetric in X and Y.

This quantity is a direct measure of the amount of information carried by X which can be learned by observing Y (or vice versa). As with entropy, there are *conditional* versions of mutual information. The mutual information between X and Y given knowledge of Z, written $\mathcal{I}(X;Y|Z)$, may be defined in a variety of ways. In particular, (6)–(8)

give rise to three equivalent definitions for $\mathcal{I}(X;Y|Z)$:

$$\mathcal{I}(X;Y|Z) = \mathcal{H}(X|Z) + \mathcal{H}(Y|Z) - \mathcal{H}(X,Y|Z) \quad (9)$$

$$\mathcal{I}(X;Y|Z) = \mathcal{H}(X|Z) - \mathcal{H}(X|Y,Z) \quad (10)$$

$$\mathcal{I}(X;Y|Z) = \mathcal{H}(Y|Z) - \mathcal{H}(Y|X,Z) \quad (11)$$

A natural definition of the leakage into X, alternative to (4), is thus:

$$\mathcal{L}'(X) = \mathcal{I}(H^\iota; X^\omega | L^\iota) \quad (12)$$

This definition is essentially the one used by [15]. In the current semantic setting, where a program defines a function from inputs to outputs, (4) and (12) are actually equivalent:

PROPOSITION 2.1. *Let X, Y, Z be random variables such that, $Z = f(X,Y)$, where f is any function. Then $\mathcal{H}(Z|Y) = \mathcal{I}(X;Z|Y)$.*

PROOF. Rewrite $\mathcal{I}(X;Z|Y)$ as $\mathcal{H}(X|Y)+\mathcal{H}(Z|Y)-\mathcal{H}(X,Z|Y)$. Then rewrite $\mathcal{H}(X|Y)$ as $\mathcal{H}(X,Y) - \mathcal{H}(Y)$, and $\mathcal{H}(X,Z|Y)$ as $\mathcal{H}(X,Z,Y) - \mathcal{H}(Y)$. Now $\mathcal{H}(X,Z,Y) = \mathcal{H}(X,Y)$, since Z is a function of (X,Y), and the result follows. □

COROLLARY 2.2. *$\mathcal{L}$ and $\mathcal{L}'$ are equivalent for a deterministic language (let $X = H^\iota$, $Y = L^\iota$, then $Z = L^\omega$ is a function of (X,Y)).*

The advantage of (12) as a definition of leakage is that it is appropriate even for a language with an inherently probabilistic semantics, whereas (4) is not. Suppose, for example, that our language contained a 'fair coin' as one of its constructs, allowing us to write:

```
X := coin;
```

Clearly, this program does not leak any information into `X`, since the final value of `X` is independent of the program's initial state. This is confirmed using (12), which gives a leakage of

$$\begin{aligned} \mathcal{I}(H^\iota; X^\omega | L^\iota) &\leq \mathcal{I}(H^\iota; X^\omega) \\ &= \mathcal{H}(H^\iota) - \mathcal{H}(H^\iota | X^\omega) \\ &= \mathcal{H}(H^\iota) - \mathcal{H}(H^\iota) = 0 \end{aligned}$$

(where we have used (7) and $\mathcal{H}(Y|Z) = \mathcal{H}(Y)$ for independent Y, Z). By contrast, applying (4) would give a leakage of

$$\mathcal{H}(X^\omega | L^\iota) = \mathcal{H}(X^\omega) = \frac{1}{2}\log 2 + \frac{1}{2}\log 2 = 1$$

Another striking difference between the current 'functional' setting and the more general probabilistic setting, is that in the functional case it is safe (conservative) to calculate a program's overall input-output leakage by considering its outputs separately:

PROPOSITION 2.3. *Let X be a vector of program variables $X_1, \ldots, X_n$ in our simple (non-probabilistic) language. Then $\mathcal{L}(X) \leq \mathcal{L}(X_1) + \cdots + \mathcal{L}(X_n)$*

PROOF. It suffices to show $\mathcal{H}(Y_1, \ldots, Y_n|Z) \leq \mathcal{H}(Y_1|Z) + \cdots + \mathcal{H}(Y_n|Z)$ for arbitrary random variables. Consider $n = 2$. Then $\mathcal{H}(Y_1, Y_2|Z) = \mathcal{H}(Y_1|Z) + \mathcal{H}(Y_2|Z) - \mathcal{I}(Y_1; Y_2|Z)$ (by Venn diagram). Since $\mathcal{I}(Y_1; Y_2|Z)$ is non-negative, the inequality holds for $n = 2$. The result follows by induction on n. □

This does *not* hold for $\mathcal{L}'$ in the setting of a probabilistic language.

2.5 Random variables and program points

Above, we used the idea that, given some distribution over the space of initial stores, the values taken by a variable X at a program's entry and exit points could be described by the pair of random variables, X^ι and X^ω.

To generalise this idea, we essentially define the random variable for X at a program point n to be such that $P(X^n = x)$ is the probability that X takes the value x *given that* control passes through program point n. To make this precise, we need to take account of two facts: firstly, n may be unreachable; secondly, for a given input store σ, control may actually pass through n many times, with X taking different values at different times. For these reasons, X^n is only defined for a subset of the program points.

Let t be a trace $(n_1, \sigma_1) \cdots (n_j, \sigma_j)$ with $n_1 = \iota$ (recall that ι is the program's entry point). We say that the trace t *decides* n_j if, for all traces which extend t, $n_i = n_j$ implies $i \leq j$. Given a program point n, let $\Delta(n)$ be the set $\{(\sigma, \sigma') | (\iota, \sigma) \cdots (n, \sigma') \text{ decides } n\}$. Note that $\Delta(n)$ is a partial function on stores (since the language is deterministic) and $\Delta(n)$ will be empty if either n is unreachable, or if all execution paths which pass through n do so infinitely often. We write $p(n)$ for the sum of the probabilities of the domain of $\Delta(n)$:

$$p(n) = \sum_{(\sigma,\sigma') \in \Delta(n)} p(\sigma)$$

$\Delta(n)$ can be interpreted as a random variable on its domain but, in general, we are interested in particular projections of $\Delta(n)$. In particular, the random variable X^n has the same domain as $\Delta(n)$ and is defined just when $p(n) > 0$:

$$P(X^n = x) = \frac{\sum_{(\sigma,\sigma') \in \Delta(n), \sigma'(X) = x} p(\sigma)}{p(n)}$$

(X may be a vector of variables, in which case $\sigma'(X)$ means the elementwise application of σ to its elements). Note that X^n describes the values taken by X immediately *before* execution of any instruction at n.

We make one further generalisation. Since we have assumed that expressions in the language define total functions on stores, and since the value of any expression is a function of the variables it contains, the definition of the random variable X^n corresponding to program variable X, automatically extends to an analogous definition of the random variable E^n, where E is any expression in the language (E^n is the random variable describing the values which E takes if evaluated at n).

2.6 Leakage at a program point

We generalise the definition of leakage above as follows. Let n be any program point; then the leakage into E at n is $\mathcal{L}^n(E) = p(n)\mathcal{H}(E^n|L^\iota)$, taking $\mathcal{L}^n(E)$ to be 0 when $p(n) = 0$. Note that, for programs which always terminate, $p(\omega) = 1$, so this generalises the previous definition with $\mathcal{L}(X) = \mathcal{L}^\omega(X)$. From sect 3.1 onwards, this paper is devoted to showing how bounds on $\mathcal{L}^n(E)$ may be calculated for each E and each n.

2.7 Perfect Secrecy, non-interference and leakage

Before detailing the analysis we show how the classical notions of non-interference can be characterised by random variable independence.

Also, although our goal is to use information theory to perform a quantitative analysis of leakage it is possible to use our definition of leakage to get qualitative results (like "the program is leaking").

We begin with a well known result by Shannon, the *Perfect Secrecy theorem*:

THEOREM 2.4. *Let X, Y, Z be random variables such that $\mathcal{I}(Z;X) = 0$ and $\mathcal{H}(Z|X,Y) = 0$. Then $\mathcal{H}(Y) \geq \mathcal{H}(Z)$.*

The result is usually interpreted in cryptography terms as saying (Z is the plain text, X the cypher text and Y the key in a secret key cryptosystem) that if the plain text and the cypher are independent ($\mathcal{I}(Z;X) = 0$) and the plain text can be recovered by key and cypher ($\mathcal{H}(Z|X,Y) = 0$) then the key has to have at least as much entropy as the plain text.

For a deterministic programming language (Z is the output, X the high input and Y low input of the program) the result says that if the output has strictly higher entropy (more information) than the low input then $\mathcal{I}(Z;X) \neq 0$, i.e. the output and the high input are not independent (in the sense of random variables).

We can think of independence of these random variables as *non-interference* of the corresponding program variables.

Given two program variables X, Y at program points $p \leq p'$ we say that X and Y *interfere* if exists an assignment of values for all remaining program variables and two values $v \neq v'$ for X at p such that the evaluation of the program at p' has Y taking the value w if $X = v$ at p and has Y taking the value $w' \neq w$ if $X = v'$ at p [1].

It is easy to see (the set of values for) Y as a function of the denotational semantics of the program fragment starting at p and ending at p'; interference means then that this map is non constant on the X component.

The m-ary definition of interference is a generalisation: $X_1, \ldots, X_m$ at program points p and Y at program point p' ($p \leq p'$) are interfering if the map (extracted from the denotational semantics as above) is non constant on the $X_1, \ldots, X_m$ components.

The concepts of independence (of random variables) and non-interference are not straightforwardly related as the following example shows:

Example: Consider the exclusive boolean or $Z = X$ `Xor` Y (true when only one of the arguments is true) with X, Y independent random variables uniformly distributed over the booleans; we have:

$$\mathcal{I}(X;Z) = \mathcal{H}(X) + \mathcal{H}(Z) - \mathcal{H}(X,Z) = 1 + 1 - 2 = 0$$

So although there is a clear interference between X and Z, this is not shown in $\mathcal{I}(X;Z)$. However if we take Y into account then interference will show:

$$\mathcal{I}(X;Z|Y) = \mathcal{H}(X|Y) + \mathcal{H}(Z|Y) - \mathcal{H}(X,Z|Y) = 0 + 1 - 0 = 1$$

The problem is that in the same way as the definition of interference refers to all (relevant) program variables (not just X and Y) we need an equivalent in the context of random variables.

The correct characterisation of non-interference for programs with multiple variables is via conditional mutual information:

[1] The definition of interference given previously in the paper is a particular case of this one.

PROPOSITION 2.5. *Let $Y, X_1, \ldots, X_n$ be random variables representing program variables with $Y = f(X_1, \ldots, X_n)$. Assume a probability distribution such that, for all $(x_1, \ldots, x_n)$, $p(X_1 = x_1, \ldots, X_n = x_n) \neq 0$. Then $Y, X_1, \ldots, X_i$ are non-interfering iff $\mathcal{I}(Y; X_1, \ldots, X_i | X_{i+1}, \ldots, X_n) = 0$.*

PROOF. The constraints on all input having non zero probability (i.e. $p(x_1, \ldots, x_n) \neq 0$) is to avoid f being a "constant in disguise" i.e. f could assume theoretically more than one value but in practice only one value is possible as the inputs for the other values have probability 0.

In the following we use X (resp. Z) for $X_1, \ldots, X_i$ (resp. $X_{i+1}, \ldots, X_n$)

From proposition 2.1 we know that $\mathcal{I}(Y; X|Z) = \mathcal{H}(Y|Z)$ so all we have to prove is that Y, X are not-interfering iff $\mathcal{H}(Y|Z) = 0$

($\Rightarrow$) : Y, X are not-interfering means the set of values for Y is a function of X, Z (i.e.$\mathcal{H}(Y|X,Z) = 0$) constant on the X component which implies $\mathcal{H}(Y|X,Z) = \mathcal{H}(Y|Z)$.

($\Leftarrow$) : Assume Y, X are interfering, i.e. $f(X,Z)$ is non constant on the X component. Then given only the Z component we will not know the value Y will assume, i.e. we will have uncertainty in Y given Z, i.e. $\mathcal{H}(Y|Z) > 0$ □

COROLLARY 2.6.
- *The definition of leakage is just a particular case of this general notion of non-interference where X and Z are the high and the low inputs.*
- *When $n = 1$ we have Y, X are non-interfering iff $\mathcal{I}(Y;X) = 0$. i.e. iff Y and X are independent random variables.*

We can now generalise Shannon result to state that if the output has strictly higher entropy (more information) than the low input then the program leaks, i.e:

THEOREM 2.7. *Let X, Y, Z random variables such that $\mathcal{I}(X;Y|Z) = 0$ and $\mathcal{H}(X|Y,Z) = 0$. Then $\mathcal{H}(Z) \geq \mathcal{H}(X)$.*

PROOF. The proof use the fact that Information Theory is a measure [16] where sum, intersection and difference are given by joint entropy, mutual information and conditional (entropy or mutual information) respectively. To view information theory as a measure allow us to reason using Venn diagrams. Consider the figure 1. The zeros correspond to $\mathcal{H}(X|Y,Z) = 0$ and $\mathcal{I}(X;Y|Z) = 0$ and $a + b = \mathcal{I}(X;Z)$. To prove that $\mathcal{H}(Z) \geq \mathcal{H}(X)$ it is hence enough to prove that $\mathcal{I}(Z;Y|X) \geq 0$ and $\mathcal{H}(Z|Y,X) \geq 0$; both these inequalities are well known basic results in information theory. □

For example consider a program whose output is a 16 bit variable and whose low input is a 8 bit variable and assume that all outputs are roughly equiprobable. Then the program leaks.

We can see from this example that the Perfect Secrecy theorem has a simple intuitive explanation: All information in the output of a deterministic program has to come from the input and if it cannot be provided by the low input then it has to be provided by the high input.

Notice that the result doesn't rely on any knowledge of the internal structure of the program; it is not possible, at this level of generality, to determine how and where the leakage occurs.

3. ANALYSING PROGRAMS FOR LEAKAGE

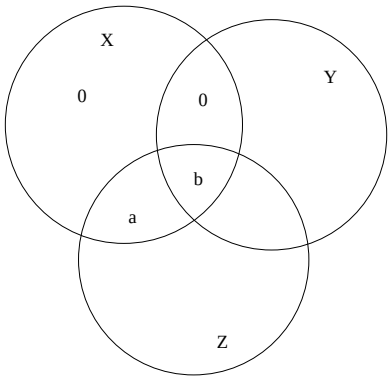


Figure 1:

This section presents the analysis, which has two main parts:

- A qualitative one (3.1–3.3): where we associate to the syntax of the program a particular graph which will summarize the information flow connections of the program.
- A quantitative one (3.4–3.7): where we provide bounds on the amount of bits leaked along this graph.

The section ends with the correctness result.

The graphs we use are Use Definition Graphs. We prefer these to the syntax because it allows us eliminate redundancy when computing leakage in loops: consider the following statements inside a loop where `high` is a leaking variable:

```
X=high; Y=Y+high;
```

In general, an assignment $Y = Y + Z$ may increase the confidential information content of Y, depending on the content of Z. In the current case, since `Y=Y+high` may be iterated, the confidential information content of `Y` may increase in a way which depends on the number of iterations. However, if `high` is not defined inside the loop, then clearly the leakage calculated for `high` should be added to that calculated for `Y` only once, since, although the assignment may be repeated, it is the *same* information which is flowing into `Y` each time. Furthermore, even if `high` is defined inside the loop, all the information which might flow through `high` must ultimately come from some collection of assignments outside the loop, allowing us to bound the maximum possible information flowing into `Y` if only we can identify the relevant set of external assignments.

It should be noted that loops are a significant hurdle in reasoning about leakage. Consider for example the Java program from sect 3.1. Assume that, at the beginning of the program, the variable `high` contains confidential information. It is easy to check that there is no direct leakage through assignments; nevertheless at each iteration one bit is leaked from `high` to `low` via an indirect flow, resulting in `low`, at the end of the loop, being a copy of `high`.

3.1 Use Definition Graphs

Given a program, the *use-definition graph* (UDG) is a directed graph whose nodes are program points.

If n is an occurrence of an assignment $X = E$ we call n a *definition* node and say that n defines X. A node n is called a *use* for the variable Y if Y appears in the expression at n (that is, the boolean expression of a control construct or the right hand side of an assignment).

There are two types of edges:

1. *data edges* ($n \longrightarrow p$): there is a data edge from n to p iff there is a non-empty path in the flowchart for the program starting from n and reaching p without any definition of X intervening and n is a definition of X or $n = \iota$, and p is a use of X or $p = \omega$;
2. *control edges* ($n \dashrightarrow p$): there is a control edge from n to p, an occurrence of $X = E$, iff n is either a `while` or an `if`-statement and p occurs inside n.

We write $\Longrightarrow$ for $\longrightarrow \cup \dashrightarrow$ and $\Longrightarrow^*$ for its transitive-reflexive closure.

3.2 UDG example

Consider the following example Java program:

```
public class Loop1 {
    public static void main(String[] args) {
        int high = Integer.parseInt(args[0]);
        int n = 16, low = 0;
        while (n >= 0) {
            int m = (int)Math.pow(2,n);
            if (high >= m) {
                low = low + m;
                high = high - m;
            }
            n = n - 1;
        }
        System.out.println(low);
    }
}
```

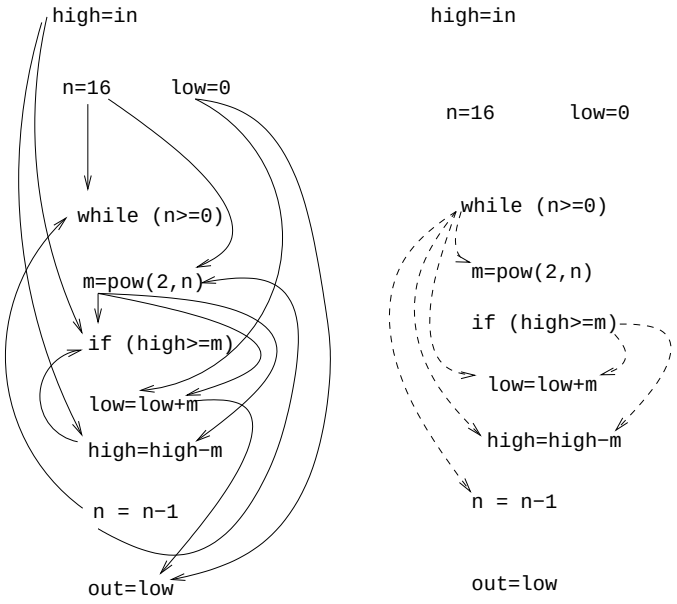
In figure 2, (a) shows the data edges and (b) shows the control edges for the essential parts of this program (we have modelled input/output as assignments from/to the distinguished variables `in`/`out`).

3.3 Source nodes

When calculating the quantity of information which has flowed into a variable at a particular program point, we use the UDG to identify the other parts of the program which make an immediate contribution; we call these the *source nodes* for the given occurrence of the variable. In defining these source nodes, we need to distinguish between those program points which lie inside a `while`-statement and those which don't; we write $n \notin W$ to mean that n does not lie within *any* `while`-statement. We need to make this distinction because of the way our analysis abstracts away from the intricacies of possible cyclic flows of information within loops: the approach within a loop (roughly speaking) is to treat a UDG *path* in the same way as an *edge* is treated outside.

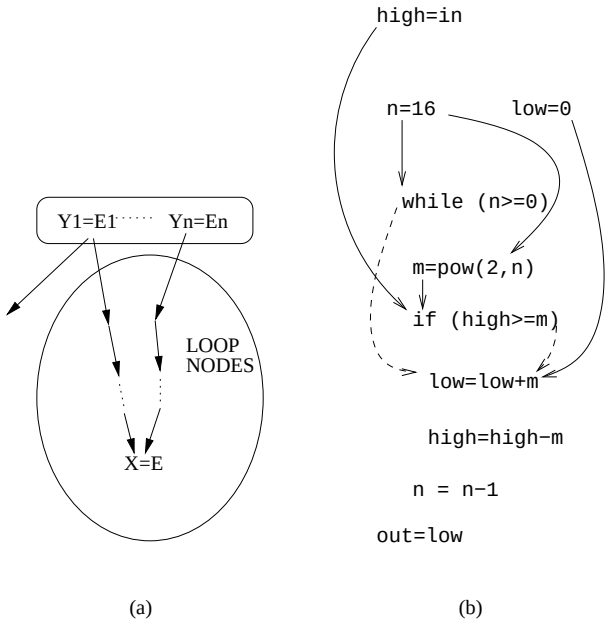
To understand the following definitions it will help to bear in mind that $a \dashrightarrow b$ iff b lies within the control structure a (`if` or `while`).

Let n be a use of the variable X; there are two types of source node, the *control source nodes for X at n*, denoted $\mathrm{con}_n(X)$, and the *data source nodes for X at n*, denoted



(a) data edges (b) control edges

Figure 2: UDG for a simple Java program



(a) (b)

Figure 3: Calculating source nodes within loops

$\mathrm{dat}_n(X)$. Wherever n occurs in the program, $\mathrm{con}_n(X)$ is the set

$$\bigcup_{m \in \mathrm{dat}_n(X)} \{m' \notin W : m' \dashrightarrow m \text{ and } m' \not\dashrightarrow n\}$$

The definition of $\mathrm{dat}_n(X)$ varies according to whether or not n lies within a `while`-statement:

1. if n lies within a `while`-statement, let w be the outermost `while` containing n, then $\mathrm{dat}_n(X)$ is the set:

$$\{m : w \not\dashrightarrow m, \exists m'.\ w \dashrightarrow m' \wedge m \longrightarrow m' \Longrightarrow^* n\}$$

2. if n does not lie within a `while`-statement, then $\mathrm{dat}_n(X)$ is the set:

$$\{m : m \text{ defines } X, m \longrightarrow n\}$$

In the definition of $\mathrm{con}_n(X)$, note that the restriction of m' to points not in any `while`-statement has the consequence that all control source nodes are `if`-statements and that no control source node lies within a `while`-statement.

Thus, the data source nodes for X at n are the assignments immediately prior to n (or to the outermost `while` containing n); the control source nodes are those `if`-statements which determine which (if any) of those assignments actually occur (assuming that control passes through n).

Where it is not necessary to distinguish between data and control source nodes, we consider the union: $\mathrm{src}_n(X) = \mathrm{dat}_n(X) \cup \mathrm{con}_n(X)$.

Each internal node in the UDG (that is, every node except and ω) has an associated expression: the right hand side for an assignment, the boolean condition for a control statement; we call this *the expression at* n, written $E(n)$. It is often necessary to consider the set of all source nodes for all the variables occurring in the expression at a node or one of its sub-expressions; given any such expression E, we denote this set $\mathrm{src}_n(E)$:

$$\mathrm{src}_n(E) = \bigcup \{\mathrm{src}_n(X) : X \text{ occurs in } E\}$$

Fig 3(a) shows a typical set of source nodes for the rhs of an assignment `X=E` at a program point p inside a loop.

From figure 3(b) we can see that the rhs of the assignment `low=low+m` inside the loop from sect 3.2 has associated the set of source nodes {`high=Integer.parseInt(args[0])`, `n=16`, `low=0`}.

3.4 Demonic attackers

Until now we have (implicitly) assumed a probability distribution on the space of initial stores which is independent of the choice of program. There are two potential problems with this assumption:

1. while it is reasonable to assume that some knowledge will be available as to the distribution of the high inputs, it is likely that little or no knowledge will be available about the low inputs;

2. the distribution for low inputs may actually be in the *control* of the attacker; in this case it would be conservative to assume that the attacker chooses L^ι to maximise leakage.

We deal with both of these problems by constructing our analysis to give results which are safe for all possible distributions on the low inputs. The approach is, essentially, to suppose that the low inputs take some fixed (but unknown) value λ. The safety of this approach is verified by proposition 3.1.

For each possible choice $L^\iota = \lambda$, we define $p_\lambda(n)$ to be the probability that program point n is eventually decided (see

sect 2.5) given that $L^\iota = \lambda$. Formally:

$$p_\lambda(n) \stackrel{\text{def}}{=} \sum_{(\sigma,\sigma')\in\Delta_\lambda(n)} p(\sigma)/P(L^\iota = \lambda),$$

where $\Delta_\lambda(n) \stackrel{\text{def}}{=} \{(\sigma,\sigma') \in \Delta(n) : \sigma(L) = \lambda\}$. Now we can define the random variables at a program point given that $L^\iota = \lambda$: just when $p_\lambda(n) > 0$, we define $X^n_\lambda \stackrel{\text{def}}{=} X^n \upharpoonright (L^\iota = \lambda)$. Finally, we can define the leakage into X at n given that $L^\iota = \lambda$: $\mathcal{L}^n_\lambda(X) \stackrel{\text{def}}{=} p_\lambda(n)\mathcal{H}(X^n_\lambda)$, taking $\mathcal{L}^n_\lambda(X) = 0$ when $p_\lambda(n) = 0$.

From here on, we assume that λ is fixed but make no assumption as to its identity. This is conservative with respect to $\mathcal{L}^n(X)$, as shown by the following:

PROPOSITION 3.1. $(\forall\lambda.\mathcal{L}^n_\lambda(X) \le a) \Rightarrow \mathcal{L}^n(X) \le a$

PROOF. By definition, $\mathcal{L}^n(X) = p(n)\mathcal{H}(X^n|L^\iota)$ and, by definition of conditional entropy, $\mathcal{H}(X^n|L^\iota)$ is the sum over all λ of $P(L^\iota = \lambda|\sigma \in D(X^n)\mathcal{H}(X^n|L^\iota = \lambda)$. The result follows simply because a weighted average is bounded by the smallest and greatest terms. □

Note that, for all $X \in L$ and for all λ, $\mathcal{L}^\iota_\lambda(X) = \mathcal{L}^\iota(X) = 0$. Furthermore, when the high-security and low-security inputs are independent, $\mathcal{L}^\iota_\lambda(Y) = \mathcal{L}^\iota(Y) = \mathcal{H}(Y^\iota)$, for all $Y \in H$.

3.5 Total versus partial random variables

The rules we present below are intended to derive bounds on the leakage into a variable at a program point, given only assumptions on the entropy of the confidential variables at the entry point. Such assumptions actually give very limited knowledge of the distribution of input values and this means that a direct calculation of the leakage at a program point is usually not possible. To illustrate, suppose program point n is the assignment `L = H` in the following program:

```
if (H < 0) then L = H else L = 1 fi
```

Now suppose that `H` and `L` are independent 32 bit variables and $\mathcal{H}(\texttt{H}) = 16$. To calculate directly the leakage into `L` we need to calculate $\mathcal{L}^n_\lambda(\texttt{H})$, but this in turn requires us to calculate both $p(n)$, which is just the probability that the condition `(H < 0)` evaluates to true, and the entropy of `H` *given that* `(H < 0)` evaluates to true. But knowing only $\mathcal{H}(\texttt{H}) = 16$ we cannot calculate either quantity. In particular, $\mathcal{H}(\texttt{H} \upharpoonright \texttt{H} < 0)$ can take essentially any value between 0 (suppose $p(h) = 0$ when $h < 0$, $p(h) \approx 1/2^{16}$ otherwise) and 32 (as before but with the condition inverted).

We deal with this difficulty by calculating bounds on the entropy of a hypothetical random variable $\hat{X}^n_\lambda$ which is defined everywhere (not just at point n) and is such that $X^n_\lambda = \hat{X}^n_\lambda \upharpoonright D(X^n_\lambda)$. The $\hat{X}^n_\lambda$ are defined in the proof of correctness (3.2).

3.6 Analysis rules

The basic analysis is given by the rules shown in table 1. For ease of exposition, we assume the following:

1. high variables $(H_1, H_2, \ldots)$ are only ever used as the rhs of assignments of the form $X = H_i$, and then at most once

2. no high variable is ever assigned

Note that these assumptions are of presentational significance only, since once H_i has been copied into X, the copy can be used and assigned freely.

A judgement $n \vdash [E] \sim a$ (where $\sim$ is one of $\le, \ge, =$) is to be read as asserting that $\mathcal{H}(\hat{E}^n_\lambda) \sim a$. In all such judgements, two things are implicit:

1. E is either $E(n)$ or a sub-expression of $E(n)$

2. E is not a high variable H_i

The initial assumptions for an analysis will be given as special *initialisation axioms* for the high variables, each of the form:

$$\frac{}{n \vdash [H_i] \sim a}$$

where n is the first and only use of H_i.

In rule [Max], bits(E) means the number of bits of storage determined by the type of the expression E (for example, if E is boolean then bits(E) is 1, whereas if E is of Java's `int` type, then bits(E) is 32).

Note that, unlike the other rules, the conclusion of rule [Low] applies only to variables, not arbitrary expressions, and applies only to program points which do not lie inside any `while`-statement. Rule [DP] is so named because it is essentially justified by the so-called Data Processing theorem ([2]) of information theory: the quantity of information output by a function cannot exceed the quantity input. Rule [Const] is really a special case of rule [DP] but is stated separately for emphasis: constant expressions transmit zero information.

As an example, consider how these rules can be applied at the statement `low=low+m` inside the loop from sect 3.1. Is is easy to see, using the rules, that we get a leakage of 16 bits (coming from `high`).

3.7 Correctness

THEOREM 3.2. *Suppose, for each initialisation axiom deriving* $n \vdash [H] \sim a$ *and for all* λ, *that* $\mathcal{H}(H^\iota_\lambda) \sim a$.

Then, for each program point n, *each sub-expression* E *of* $E(n)$, *and each* λ, *there exists a random variable* $\hat{E}^n_\lambda$ *such that:*

1. $E^n_\lambda = \hat{E}^n_\lambda \upharpoonright D(E^n_\lambda)$

2. *whenever the rules in table 1 derive* $n \vdash [E] \sim a$, *then* $\mathcal{H}(\hat{E}^n_\lambda) \sim a$

COROLLARY 3.3. $n \vdash [E] \le b$ *implies* $\mathcal{L}^n(E) \le b$.

To see why the corollary follows, first apply proposition 3.1, giving $\mathcal{H}(\hat{E}^n|L^\iota) \le b$, let B be the characteristic function for $D(E^n)$ and let $q = P(B = t)$. Then we have $\mathcal{H}(\hat{E}^n|L^\iota, B) = q\mathcal{H}(E^n|L^\iota) + (1-q)\mathcal{H}(\hat{E}^n \upharpoonright (B = f)|L^\iota)$. Thus $\mathcal{L}^n(E) = q\mathcal{H}(E^n|L^\iota) \le \mathcal{H}(\hat{E}^n|L^\iota, B) \le \mathcal{H}(\hat{E}^n|L^\iota) \le b$.

The rest of this section sketches a proof of the theorem.

First we must explain how $\hat{E}^n$ can be constructed. The key observation is that these functions actually arise naturally in a *denotational* semantics. Consider the program P:

```
A=B; if (B) then X=Y;M=N; else Z=W fi
```

$$[\text{Min}]\ \frac{}{n \vdash [E] \geq 0} \qquad [\text{Max}]\ \frac{}{n \vdash [E] \leq \text{bits}(E)}$$

$$[\text{DP}]\ \frac{n_1 \vdash [E(n_1)] \leq b_1, \ldots, n_k \vdash [E(n_k)] \leq b_k}{n \vdash [E] \leq \sum_{i=1}^{k} b_k}\ \text{src}_n(E) = \{n_1, \ldots, n_k\}$$

$$[\text{Const}]\ \frac{}{n \vdash [E] = 0}\ \text{src}_n(E) = \emptyset$$

$$[\text{Low}]\ \frac{p \vdash [E(p)] \geq a}{n \vdash [X] \geq a}\ n \notin W, \text{src}_n(X) = \{p\}$$

Table 1: Analysis rules

Denotationally, we have

$$[\![P]\!] = (\lambda\sigma.\text{cond}([\![\texttt{B}]\!]\sigma, [\![\texttt{X} = \texttt{Y}; \texttt{M} = \texttt{N}]\!]\sigma, [\![\texttt{Z} = \texttt{W}]\!]\sigma)) \circ [\![\texttt{A} = \texttt{B}]\!].$$

Note that this is equivalent to

$$\lambda\sigma.\text{cond}([\![\texttt{B}]\!]\sigma, [\![\texttt{A} = \texttt{B}; \texttt{X} = \texttt{Y}; \texttt{M} = \texttt{N}]\!]\sigma, [\![\texttt{A} = \texttt{B}; \texttt{Z} = \texttt{W}]\!]\sigma).$$

Now let n be the program point `M=N`. In this example, $\hat{N}^n$ is the total function $[\![\texttt{A} = \texttt{B}; \texttt{X} = \texttt{Y}]\!]$ whereas N^n is the partial function $[\![\texttt{A} = \texttt{B}; \texttt{X} = \texttt{Y}]\!]$ restricted to those σ such that $[\![\texttt{A} = \texttt{B}]\!]\sigma = t$. Clearly it is not quite straightforward to generalise this idea to `while`-statements. However, our analysis treats loops in such a way that it is only necessary to establish the *existence* of a total function for each point p inside a loop which extends the partial function E^p. This is easily seen to be the case.

The remaining work of the proof is to establish that the UDG structure safely over approximates all possible dependencies among the $\hat{E}n$. It is then possible to show that each $\hat{E}^p$ can be expressed as a function of the $\hat{E}^n$ corresponding to the source nodes for p. Then we can apply the following lemma:

LEMMA 3.4 (DATA PROCESSING). *Let $X_1, \ldots, X_n, Z$ be random variables with a common domain D and such that $Z = f(X_1, \ldots, X_n)$, where f is any total function on D. Then $\mathcal{H}(Z) \leq \mathcal{H}(X_1, \ldots, X_n)$.*

As an immediate corollary, $\mathcal{H}(Z) \leq \mathcal{H}(X_1) + \cdots + \mathcal{H}(X_n)$, since $\mathcal{H}(X_1, \ldots, X_n) \leq \mathcal{H}(X_1) + \cdots + \mathcal{H}(X_n)$.

In particular, the $\hat{E}^n$ for the control source nodes play the role of B in the following results:

LEMMA 3.5 (DOMAIN PARTITION). *Let X be any random variable and let B be a boolean-valued random variable (range $\{t, f\}$) and let $q = P(B = t)$. Writing X_B to mean $X \upharpoonright (B = t)$ and $X_{\neg B}$ to mean $X \upharpoonright (B = \text{false})$:*

$$\mathcal{H}(X) \leq q\mathcal{H}(X_B) + (1 - q)\mathcal{H}(X_{\neg B}) + \mathcal{H}(B)$$

PROOF. Note that $\mathcal{H}(X) \leq \mathcal{H}(H, B) = \mathcal{H}(X|B) + \mathcal{H}(B)$. The result is then immediate by definition of conditional entropy (equation 3). □

Now, given two random variables X_1, X_2 with range R, and a boolean-valued random variable B, we write $X_1 \oplus_B X_2$ to mean the random variable with range R and defined as follows:

$$X_1 \oplus_B X_2 = \begin{cases} X_1 & \text{when } B = t \\ X_2 & \text{when } B = f \end{cases}$$

It is easily seen that $P((X_1 \oplus_B X_2) = x) = P(B = t)P(X_1 = x|B = t) + P(B = f)P(X_2 = x|B = f)$.

PROPOSITION 3.6. $\mathcal{H}(X_1 \oplus_B X_2) \leq P(B = t)\mathcal{H}(X_1 \upharpoonright B = t) + P(B = f)\mathcal{H}(X_2 \upharpoonright B = f) + \mathcal{H}(B)$

PROOF. Note that $(X_1 \oplus_B X_2) \upharpoonright (B = t) = X_1 \upharpoonright (B = t)$ and $(X_1 \oplus_B X_2) \upharpoonright (B = f) = X_2 \upharpoonright (B = f)$, then apply the Domain Partition lemma. □

4. REFINING THE ANALYSIS OF EXPRESSIONS

The rules in table 1 allow us to measure flows of information through the program but in a crude way, analogous to plumbing pipes together. The utility of an implemented analysis based on our approach will depend on its sensitivity, i.e. on being able to establish tight as opposed to loose bounds on the leakage of information. Although we are limited to the basic analysis rules inside loops, outside loops it is sometimes possible to improve the sensitivity by substituting a more refined rule which uses knowledge of the expression being analysed.

The discovery of these refined rules is an on-going project. In the remainder of this section we present some results for a more refined analysis of equality tests and some arithmetic operations. The resulting rules may be found in table 2.

4.1 Analysis of equality tests

In this section we develop a refined rule for analysis of tests of the form $E_1 == E_2$. Our development is motivated by a simple observation: when the distribution of values for E_1 is close to uniform (high entropy) and the distribution for E_2 is concentrated on just a few values (low entropy), then *most of the time*, E_1 and E_2 will not be equal.

We can quantify the implications of this observation by considering the following question: suppose that X is a k-bit random variable and suppose that $P(X = x) = q$, for some q; what is the maximum possible value for $\mathcal{H}(X)$? We call this maximum the *upper entropy for q in k bits*, denoted $\mathcal{U}_k(q)$. Since entropy is maximised by uniform distributions, the maximum value possible for $\mathcal{H}(V)$ is obtained in the case that $P(X = x')$ is uniformly distributed for all $x' \neq x$. There are $2^k - 1$ such x' and applying the definition of H (eqn. 1) immediately gives:

$$\mathcal{U}_k(q) \stackrel{\text{def}}{=} q \log \frac{1}{q} + (1 - q) \log \frac{2^k - 1}{1 - q} \tag{13}$$

As the following proposition shows, if $P(X = Y) = q$, then $\mathcal{U}_k(q)$ is an upper bound for the *difference* between $\mathcal{H}(X)$ and $\mathcal{H}(Y)$.

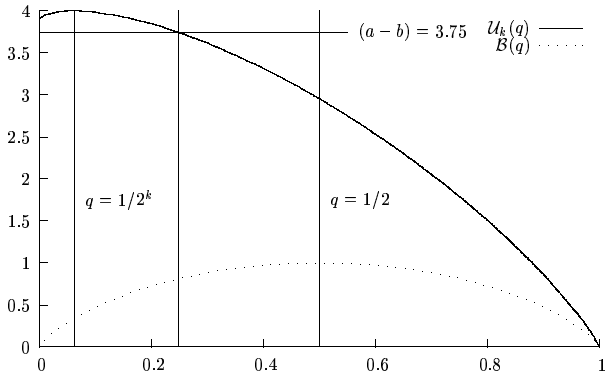


Figure 4: the upper entropy for q in 4 bits

PROPOSITION 4.1. *Given a k-bit random variable X and any other random variable Y, let $q = P(X = Y)$ (by $P(X = Y)$ we mean $\sum_{x=y} p(x, y)$). Then*

$$\mathcal{H}(X|Y) \leq \mathcal{U}_k(q)$$

PROOF. □

As an immediate corollary we have $\mathcal{H}(X) - \mathcal{H}(Y) \leq \mathcal{U}_k(q)$, since $\mathcal{H}(X|Y) = \mathcal{H}(X, Y) - \mathcal{H}(Y) \geq \mathcal{H}(X) - \mathcal{H}(Y)$.

Now the quantity of interest ($\mathcal{H}(X{=}{=}Y)$) is just $\mathcal{B}(q)$, where

$$\mathcal{B}(q) \stackrel{\text{def}}{=} q \log \frac{1}{q} + (1-q) \log \frac{1}{1-q} \tag{14}$$

It is easily seen that $\mathcal{B}(q)$ is an increasing function of q in the region $0 \leq q \leq 0.5$ and this is sufficient to justify the refined rule [Eq] for equality tests (see table 2).

(We leave unstated the companion rule, justified by commutativity of $+$, which reverses the roles of E_1 and E_2.) We note that [Eq] will give useful results in the case that a is high and b is low, that is, when E_1 is known to contain a large amount of confidential information and E_2 is known to contain very little.

The way in which rule [Eq] can be applied is illustrated by the example shown in fig. 4. This plots $\mathcal{U}_k(q)$ and $\mathcal{B}(q)$ against q for $k = 4$ and shows that for a lower bound of $(a - b) = 3.75$, q is bounded by $0 \leq q \leq 0.25$ (the precise upper bound is slightly lower than this). To find a maximum for q, we need to solve equations of the form $\mathcal{U}_k(q) - (a-b) = 0$ and, for this, simple numerical techniques suffice [6]. (Note also that $\mathcal{B}(q) + (1-q)k$ is an upper bound for $\mathcal{U}_k(q)$ and that this bound is very tight unless k is small.)

4.1.1 Example

Consider the program P:

```
Y = H; if (Y == 0) then X = 0 else X = 1 fi; Z = X!
```

with `H` high-security. Let program points p, n_0 and n_1 be the `if`-statement, the assignment `X = 0`, the assignment `X = 1`, and the assignment `Z = X`, respectively. Suppose that $k = 32$ and the input distribution makes H uniform over its 2^{32} possible values. Thus we can analyse the program starting with the assumption $\iota \vdash [H] \geq 32$. The basic rules plus [Eq] are then easily seen to derive:

$$p \vdash [\texttt{Y} == 0] \leq \epsilon$$

where $\epsilon = \mathcal{B}(1/2^{32}) \approx 7.8 \times 10^{-7}$. Thus, using [Const] and [DP], we derive

$$n \vdash [\texttt{X}] \leq \epsilon$$

5. ANALYSIS OF ARITHMETIC EXPRESSIONS

We can improve the analysis of leakage via arithmetic expressions by exploiting algebraic knowledge of the operations together with information about the operands acquired through supplementary analyses such as parity analysis, constant propagation analysis etc. The (binary) operations we consider are addition, subtraction, multiplication $(+, -, *)$ on the twos-complement representations of k bit integers with overflow.

We use $\odot$ for the binary operator while the random variables X, Y and Z range over the first and second inputs and the output of the operator respectively. They are related by

$$P(Z = z) = \sum_{(x,y) \in \odot^{-1}(z)} P(X = x, Y = y)$$

We assume we know bounds on the entropy of the input space, $\mathcal{H}(X, Y)$, and entropy of the projected input spaces, $\mathcal{H}(X)$ and $\mathcal{H}(Y)$ and we aim to find bounds on the entropy of the output space, $\mathcal{H}(Z)$.

Since a binary arithmetic operation on twos complement integers is a function from $X \times Y$ to Z and since functions can only reduce entropy or leave it the same we have

$$0 \leq \mathcal{H}(Z) \leq \mathcal{H}(X, Y)$$

In general we will not know $\mathcal{H}(X, Y)$ but only $\mathcal{H}(X)$ and $\mathcal{H}(Y)$. Since $\mathcal{H}(X, Y) \leq \mathcal{H}(X) + \mathcal{H}(Y)$ we can use this sum as an upper bound but we can then only deduce as much as can be discovered using two of the default rules for the analysis, [Min] and [DP]. The rule [Min] always applies. The upper bound observation is captured for an operation $\odot$ in the rule [Opmax] in table 2.

Further improvements to either the upper or the lower bound depend on knowledge more specific to the operation and/or the expressions.

Before examining individual operations there is something we can say about the relationship between $\mathcal{H}(Z)$ and $\mathcal{H}(X, Y)$ that holds for all operations which we exploit directly in proposition 5.3.

We can use the functional relationship between inputs and outputs to show that the entropy of the output space is the entropy of the input space less the entropy of the input space *given knowledge of the output space*. This latter quantity is a measure of the entropy of the input space destroyed by the function $\odot$. The idea is expressed formally in the following proposition:

PROPOSITION 5.1. *Let $Z = \odot(X, Y)$ then $\mathcal{H}(Z) = \mathcal{H}(X, Y) - \mathcal{H}(X, Y|Z)$*

PROOF. Using conditional entropy expressions we have this relationship between $\mathcal{H}(X, Y)$ and $\mathcal{H}(Z)$: $\mathcal{H}(X, Y, Z) = \mathcal{H}(Z) + \mathcal{H}(X, Y|Z)$. However we also have

$$\begin{aligned} p(x, y, z) &= p(x, y) \text{ if } (x, y) \in \odot^{-1}(z) \\ &= 0 \text{ otherwise} \end{aligned}$$

$$[\text{Eq}]\ \frac{n \vdash [E_1] \geq a \quad n \vdash [E_2] \leq b}{n \vdash [E_1 == E_2] \leq \mathcal{B}(q)}\ q \leq 0.5, \mathcal{U}_k(q) \geq (a-b), k = \text{bits}(E_1)$$

$$[\text{Opmax}]\ \frac{n \vdash [E_1] \leq b_1\ n \vdash [E_2] \leq b_2}{n \vdash [E_1 \odot E_2] \leq b_1 + b_2}$$

$$[\text{AddMin}]\ \frac{n \vdash [E_1] \geq a_1\ n \vdash [E_2] \geq a_2\ n \vdash [E_1] \leq b_1\ n \vdash [E_2] \leq b_2}{n \vdash [E_1 + E_2] \geq \max(a_1, a_2)\ -\ \min(b_1, b_2)}$$

$$[\text{Zeromult}]\ \frac{n \vdash E_1 \text{ is } 0\ \vee\ n \vdash E_2 \text{ is } 0}{n \vdash [E_1 * E_2] = 0}$$

$$[\text{Oddmult}]\ \frac{n \vdash E_1 \text{ is odd constant}\ \ n \vdash [E_2] \geq b_2}{n \vdash [E_1 * E_2] \geq n \vdash [E_2]}$$

Table 2: Some Refined Analysis rules

Hence

$$\begin{aligned}\mathcal{H}(X,Y,Z) &= -\sum_{x,y,z} p(x,y,z)\log(p(x,y,z)) \\ &= -\sum_{x,y} p(x,y)\log(p(x,y)) \\ &= \mathcal{H}(x,y)\end{aligned}$$

Then we have $\mathcal{H}(X,Y) = \mathcal{H}(Z) + \mathcal{H}(X,Y|Z)$ and so $\mathcal{H}(Z) = \mathcal{H}(X,Y) - \mathcal{H}(X,Y|Z)$ □

5.1 Addition and subtraction

Addition and subtraction are the same operation in twos complement arithmetic so we restrict our attention to addition.

Bitwise addition (+) makes the set of numbers representable in twos-complement using k bits a cyclic additive group with identity 0 and generator 1 (so $a+1$ has its usual meaning except when $a = 2^{k-1}-1$, in which case $a+1 = -2^{k-1}$). The inverse $-a$ is given by the twos-complement operation (so $-a$ has its usual meaning except for $a = -2^{k-1}$, which is its own inverse).

PROPOSITION 5.2. *Let $T_k = \{-2^{k-1}, \ldots, 2^{k-1}-1\}$, the set of integers representable by k bits in twos complement. Bitwise addition (+) makes T_k a cyclic additive group with identity 0 and generator 1.*

We don't include a proof here but the proposition is straightforward to verify.

The advantage of this knowledge of the operation structure is that it allows us to show, for addition, either operand is a function of both the other operand and the result of the operation. This in turn allows us to establish a tighter lower bound for this operation: The entropy of the outcome, $\mathcal{H}(Z)$, is bigger than or equal to the entropy of the input space, $\mathcal{H}(X,Y)$ less the smaller of the two projected entropies for that space, $\mathcal{H}(X)$ and $\mathcal{H}(Y)$.

PROPOSITION 5.3. *Let $Z = +(X,Y)$, then*

$$\mathcal{H}(Z) \geq \mathcal{H}(X,Y) - min(\mathcal{H}(X), \mathcal{H}(Y))$$

PROOF. First we establish that $\mathcal{H}(Z) \geq \mathcal{H}(X,Y) - \mathcal{H}(X)$.

We can make arguments similar to the one employed to justify proposition 5.1 and demonstrate that $\mathcal{H}(X,Y,Z) = \mathcal{H}(X,Z)$.

By proposition 5.1 we have $\mathcal{H}(Z) = \mathcal{H}(X,Y) - \mathcal{H}(X,Y|Z)$ so it suffices to show that $\mathcal{H}(X,Y|Z) \leq \mathcal{H}(X)$

$$\begin{aligned}&\mathcal{H}((X,Y|Z) \\ =\ &\mathcal{H}(X,Y,Z) - \mathcal{H}(Z) \\ =\ &\mathcal{H}(X,Z) - \mathcal{H}(Z) \\ =\ &\mathcal{H}(X|Z) \\ \leq\ &\mathcal{H}(X)\end{aligned}$$

By a similar argument we can also establish $\mathcal{H}(Z) \geq \mathcal{H}(X,Y) - \mathcal{H}(Y)$. These two inequalities establish the proposition. □

Since $\mathcal{H}(X,Y) \geq \max(\mathcal{H}(X), \mathcal{H}(Y))$ we can safely replace $\mathcal{H}(X,Y)$ with that quantity. This provides the rule [AddMin] in table 2 for calculating an improved lower bound for addition.

5.2 Multiplication

Multiplication is less straightforward to analyse than addition as the algebraic structure of the operation is more complex. We are not currently able to provide any general result for the operation. However, in the event that some subsidiary prior analysis is able to identify a useful property of one of the operands of the operation, we can get very good bounds on the entropy of the output, in particular when one of the operands is odd or when one of the operands is zero.

Multiplication by zero always has zero as the result, i.e. Z has value space singleton set $\{0\}$ whose element has probability 1, so knowing that one operand is zero guarantees that $\mathcal{H}(Z) = 0$. This observation is captured in the rule [Zeromult] in table 2. .

To understand why it is useful to learn that one operand for multiplication is odd, consider the following. For a in T_k and $n \geq 0$, we can write $n.a$ to mean $a + \cdots + a$ (n times), taking $0.a = 0$. Multiplication in twos-complement arithmetic is defined such that it distributes over addition and $a * 1 = a$. It follows that $a * (n.1) = n.a$. Note that every b in T_k can be written as $n.1$ for some (unique) n, $0 \leq n < 2^k$, and thus $a * b$ can be understood instead as $n.a$.

Recall that the order of a group is the number of its elements, hence the order of T_k is 2^k. Furthermore, the order of any element a is defined to be the order of the cyclic sub-

group $<a>$ which it generates (with elements $\{n.a : n \geq 0\}$). We denote the order of an element a by $o(a)$. We can then state the following

PROPOSITION 5.4. *For $a \in T_k$ where a is odd, $o(a) = 2^k$, i.e. any odd element is a generator for T_k.*

This can be demonstrated using some elementary group theory since every odd number does not divide the order of T_k.

Because odd elements are generators for the whole set, multiplication by an odd constant (i.e. zero entropy in one component) can be viewed as an injective function from T_k to T_k and so the entropy of the output space can never be less than the entropy of the other operand.

PROPOSITION 5.5. *Let $Z = *(X, Y)$ where $\forall x \in X$. x is an odd constant, then*

$$\mathcal{H}(Z) \geq \mathcal{H}(Y)$$

.

The implication of this result for propagation of lower bounds during multiplication is captured in the rule [Oddmult] in table 2. The rule is also intended to represent its equivalent rule after applying commutivity of multiplication.

Although we don't have anything we can say in general about leakage via multiplication we do have a theoretical result which may prove useful in future work: we know the size of the inverse image of an element of T_k.

THEOREM 5.6. *For c in T_k,*

$$|*^{-1}(c)| = \begin{cases} (k + 1 - \log o(c))2^{k-1} & \text{if } c \neq 0 \\ (k+2)2^{k-1} & \text{if } c = 0 \end{cases}$$

6. IMPROVEMENTS TO THE ANALYSIS

Our analysis of `if`-statements is effectively distributed between the definition of the UDG structure and the rule [DP]. As shown by the proof of correctness (theorem 3.2), the essential principle is that, at a point n immediately following an `if`-statement in which X may be defined, $\hat{X}_lambda^n$ can be viewed as function of B, Y and Z, corresponding to the condition and the values of X at the end of the true and false branches, respectively.s The DP lemma then implies that $\mathcal{H}(\hat{X}_\lambda^n) \leq \mathcal{H}(B) + \mathcal{H}(Y) + \mathcal{H}(Z)$. The weakness of this approach is that it takes no account of the relative probabilities of either branch being chosen. Bounds on the probabilities will in many cases be available (provided, for example, by the analysis of equality tests). As an example let P' be the program

```
Y = H; if (Y==0) then X = Y else X = 1 fi; Z = X
```

This is semantically equivalent to P in sect. 4.1.1 but the best we can derive for P' is the totally uninformative:

$$n \vdash [\mathtt{Z}] \leq 32$$

The problem is caused by the statement `X = Y`. In isolation this would leak all the information from `Y` into `X` but, in the context of this `if`-statement, it actually leaks *no* information.

We can improve on such examples by using what we know about q, where q is the probability that the condition evaluates to true. For equality tests (see sect 4.1) we may have an explicit bound on q as a result of applying the [Eq] rule. More generally, given $n \vdash [B] \leq b$ for a boolean expression B, we can invert $\mathcal{B}(q)$ to find an upper bound on q or $1 - q$ (though in this case, we don't know which). Proposition 3.6 can then be used to tighten the upper bound derived in some cases. In the example above, application of the [Eq] rule provides a bound on q of $1/2^{32}$. Since we know that $\mathcal{H}(X \restriction (B = t)) \leq k$ for any k-bit variable X, proposition 3.6 allows us to derive the much improved:

$$n \vdash [\mathtt{Z}] \leq 32/2^{32} + \mathcal{B}(1/2^{32}) \approx 1.5 \times 10^{-8}$$

7. CONCLUSIONS AND FUTURE WORK

The work presented in this paper is the first time Information Theory has been used to measure interference between variables in a simple imperative language. An obvious and very desirable extension of the work would be to a language with probabilistic operators.

Incremental improvement of the analysis could be given by a subtler treatment of loops and by improved bounds on a wider range of expressions. A similar syntax based, security related analysis might be applied to queries on a secure database. Denning [3] did work on information flow in database queries.

It would be also interesting to be able to provide a"backward" analysis where given an interference property that we want a particular program point to satisfy we are able to deduce constraints on the interference of the input. A simple example of this scenario is provided by Shannon's perfect secrecy theorem where the property of non-interference on the output implies the inequality $\mathcal{H}(L) \geq \mathcal{H}(\mathtt{Out})$

Timing issues like "rate of interference" could also be analysed by our theory allowing for a quantitative analysis of "timing attacks" [14]..

On a more speculative level we hope that quantified interference could play a role in fields where modularity is an issue (for example model checking or information hiding). At the present modular reasoning seems to break down whenever modules interfere by means other than their interfaces. However if once quantified the interference is shown to be below the threshold that affects the desired behaviour of the modules, it could be possible to still use modular reasoning. An interesting development in this respect could be to investigate the integration of quantified interference with non-interference based logic [9, 5].

The authors would like to thank Peter O'Hearn for helpful comments on this work.

8. REFERENCES

[1] D. Clark, S. Hunt, and P. Malacaria. Quantitative analysis of the leakage of confidential data. In A. D. Pierro and H. Wiklicky, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier, 2002.

[2] T. M. Cover and J. A. Thomas. *Elements of Information Theory.* Wiley Interscience, 1991.

[3] D. E. R. Denning. *Cryptography and Data Security.* Addison-Wesley, 1982.

[4] J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, 1982.

[5] S. Isthiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *28th POPL*, pages 14–26, London, January 2001.

[6] R. L.Burden and J. D. Faires. *Numerical Analysis*. PWS-KENT, 1989. ISBN 0-534-93219-3.

[7] J. Millen. Covert channel capacity. In *Proc. 1987 IEEE Symposium on Research in Security and Privacy*. IEEE Computer Society Press, 1987.

[8] A. D. Pierro, C. Hankin, and H. Wiklicky. Approximate non-interference. In I. Cervesato, editor, *CSFW'02 – 15th IEEE Computer Security Foundation Workshop*. IEEE Computer Society Press, June 2002.

[9] J. Reynolds. Separation logic: a logic for shared mutable data structures. Invited Paper, LICS'02, 2002.

[10] J. C. Reynolds. Syntactic control of interference. In *Conf. Record 5th ACM Symp. on Principles of Programming Languages*, pages 39–46, Tucson, Arizona, 1978. ACM, New York.

[11] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. In *Proc. European Symposium on Programming*, Amsterdam, The Netherlands, March 1999. ACM Press.

[12] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. 13th IEEE Computer Security Foundations Workshop*, Cambridge, England, July 2000. IEEE Computer Society Press.

[13] C. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423 and 623–656, July and October 1948. Available on-line at http://cm.bell-labs.com/cm/ms/what/shannonday/paper.html.

[14] D. Volpano. Safety versus secrecy. In *Proc. 6th Int'l Symposium on Static Analysis*, LNCS, pages 303–311, Sep 1999.

[15] J. W. Gray, III. Toward a mathematical foundation for information flow security. In *Proc. 1991 IEEE Symposium on Security and Privacy*, pages 21–34, Oakland, CA, May 1991.

[16] R. W. Yeung. A new outlook on shannon's information measures. *IEEE Transactions on Information Theory*, 37(3), May 1991.