Quantified Interference for a While Language

David Clark [1]

Department of Computer Science
King's College London
London, UK

Sebastian Hunt [2]

Department of Computing
City University
London, UK

Pasquale Malacaria [3]

Department of Computer Science
Queen Mary College, University of London
London, 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 in While language programs. 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.

Key words: security, program analysis, information theory, information flow, interference

1 Introduction

Quantifying interference is an important part of assessing covert channels in security devices, and doing this is less well-established for programmable components than for simple hardware components.

[1] Email: david@dcs.kcl.ac.uk
[2] Email: seb@soi.city.ac.uk
[3] Email: pm@dcs.qmul.ac.uk

Take the notion of *interference* [5,10] 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-behaved, whereas interference can lead to mysterious (mis-)behaviours. However, significant misbehaviours caused by interference will generally happen only when there is *enough* interference. 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 [5] so they are not "secure" in a qualitative sense. However, common sense suggests to consider them secure if the interference is very small. This paper uses Shannon's information theory [13] to define a quantified notion of interference for a simple imperative language and derives a program analysis based on this notion.

We briefly recall Shannon's key definitions and use them to define a quantitative measure of the leakage into each variable at each program point for a While language. We then use these definitions as the basis for a program analysis which derives 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:

Analysis structure and loops: The analysis is now graph and not syntax based; this allows us to handle loops.

Equality tests: we analyse general equality tests (not just tests against a constant, as previously).

Arithmetic expressions: we present a significantly improved analysis of arithmetic operators which exploits their algebraic properties.

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 [4] 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 Millen [8] 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 Gray [14], which develops a 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 [9]. Their interest has been to measure interference in the context of a probabilistic concurrent constraint setting where the interference comes via probabilistic operators. 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 work which develops an automatable analysis for the *quantity* of information (in Shannon's sense) which may be leaked by a program.

A paper on confidentiality properties which has recently come to the authors' attention, and which does use information theory, is [7]. Their definition of 'information escape' is similar to our definition of leakage (sect 2.3), though it does not appear to be equivalent. The focus of [7] is on the relationship between security properties and refinement, rather than analysing the quantity of information leaked by a program. The connections with the work of the current paper deserve further investigation.

2 Information theory and leakage analysis

2.1 The language

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 a partial 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.

We note that the primary interest in dealing with loops is not the possibility of non-termination but the potential for an arbitrary quantity of information to be leaked via iterations containing implicit information flows (an implicit flow being one which is achieved via control flow rather than assignment from confidential sources [4]).

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 $\rightarrow$ 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 $(n_i, \sigma_i) \rightarrow (n_{i+1}, \sigma_{i+1})$ for $1 \leq i < j$.

2.2 Interference and information theory

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 [5], is based on the notion of *non-interference*, in our setting: whether or not a program leaks confidential information. Here by contrast we address the question of *how much* may be leaked.

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. For a discussion of the information theoretic background to the current work see our earlier papers [1,2]. Recall that Shannon's theory is based on the fundamental quantity $\mathcal{H}$, variously known as *information* or *entropy*, defined thus:

$$\mathcal{H}(X) \stackrel{\text{def}}{=} \sum_x p(x) \log \frac{1}{p(x)} \qquad (1)$$

where X is a random variable, $p(x)$ is shorthand for $P(X = x)$, the probability that random variable $X = x$, and the sum is over the range of X. The derived quantity of *conditional entropy* is $\mathcal{H}(X|Y) \stackrel{\text{def}}{=} \sum_y p(y)\mathcal{H}(X|Y = y)$, where $\mathcal{H}(X|Y = y) \stackrel{\text{def}}{=} \sum_x p(x) \log \frac{1}{p(x|y)}$ and $p(x|y)$ is the probability that the random variable $X = x$ given that random variable $Y = y$.

2.3 Random variables and program points

We define for each program point a random variable corresponding to observations of the value of the variable at this point. In particular, we are interested in how much of the information carried by the high inputs to a program can be learnt 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 (assume termination for now; this is relaxed below). We take as a measure of the amount of leakage into X due to the program:

$$\mathcal{L}(X) \stackrel{\text{def}}{=} \mathcal{H}(X^\omega | L^\iota) \tag{2}$$

(where L^ι is the random variable describing the distribution of the program's non-confidential inputs). We note that non-interference in this setting is just the special case of 0 leakage. (We give a precise formal presentation of this result, and more general discussion of why the definition is appropriate elsewhere, see [2].)

The random variable for X at a program point n is defined 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. However n may be unreachable or 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 $n_1 = \iota$. 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 $\{(\sigma, \sigma') | (\iota, \sigma) \cdots (n, \sigma') \text{ decides } n\}$. We write $p(n)$ for the sum of the probabilities of the domain of $\Delta(n)$:

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

$\Delta(n)$ 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 $p(n) > 0$:

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

(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. The definition of random variables X^n automatically extends to an analogous definition of random variables 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).

We generalise the definition of leakage above (equation 2) as follows. Let n be any program point. Then the leakage into E at n is $\mathcal{L}^n(E) \stackrel{\text{def}}{=} p(n)\mathcal{H}(E^n | L^\iota)$, 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 $\mathcal{L}(X) \stackrel{\text{def}}{=} \mathcal{L}^\omega(X)$. The remainder of this paper is devoted to showing how bounds on $\mathcal{L}^n(E)$ may be calculated.

3 Analysing programs for leakage

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

- A qualitative one (3.1–3.2): where we associate to the syntax of a program a graph summarising its information flow connections.
- A quantitative one (3.3–3.6): where we provide bounds on the amount (number of bits) of information leaked along this graph.

The section ends with the correctness result.

The graphs we use are a form of use-definition graph. We prefer these to the syntax because they expose the structure we use to deal with loops. It should be noted that loops are a significant hurdle in the analysis of leakage. Consider the program in sect 3.1. A little thought shows that, for `in` taking values in $0 \leq i < 2^{16} - 1$, this program copies `in` to `out` via the variables `high` and `low`, even though there is no assignment to `low` from `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:

(i) *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 $p = \omega$;

(ii) *control edges* ($n \dashrightarrow p$): there is a control edge from n to p iff n is either a `while` or an `if`-statement and p is an assignment which occurs inside that statement.

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

Consider the following example program:

```
int high = in;
int b = 15, low = 0;
while (b >= 0) {
    int m = (int)Math.pow(2,b);
    if (high >= m) {
        low = low + m; high = high - m;
    }
    b = b - 1;
}
out = low;
```

Figure 1, shows the data edges (a) and control edges (b) for this program.

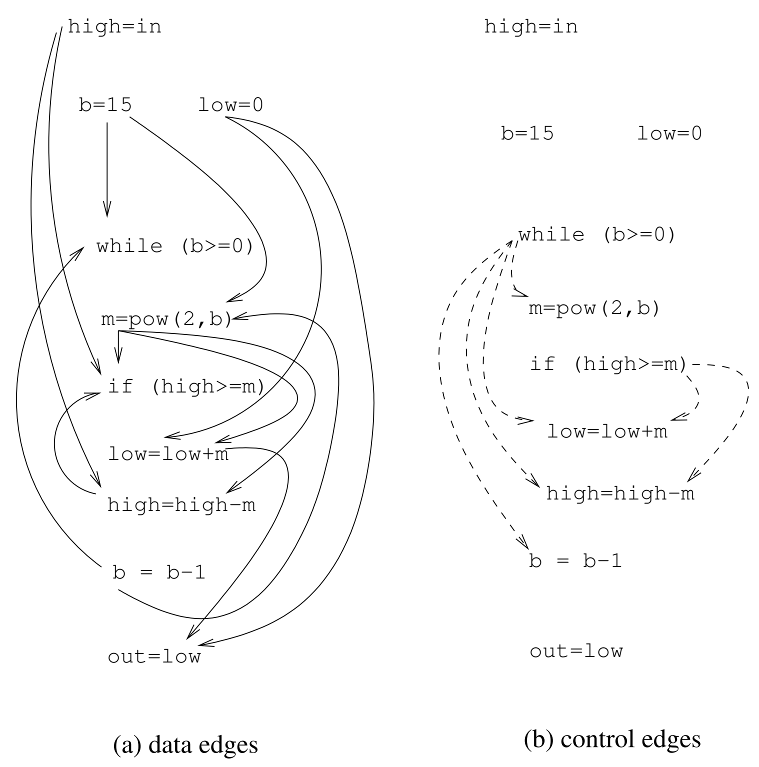

Fig. 1. UDG for a simple Java program

3.2 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 $\mathrm{dat}_n(X)$. 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:

(i) If n lies within a `while`-statement, let w be the outermost `while` containing n, then $\mathrm{dat}_n(X)$ is the set: $\{m : w \not\dashrightarrow m, \exists m'.\ w \dashrightarrow m' \wedge m \longrightarrow m' \Longrightarrow^* n\}$.

(ii) 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. This is not because `while` conditions cannot influence control flow (they clearly do) but because our analysis of loops is more pessimistic than our analysis of `if`-statements.

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: $\mathrm{src}_n(X) \overset{\text{def}}{=} \mathrm{dat}_n(X) \cup \mathrm{con}_n(X)$.

Each internal node in the UDG (that is, every node except ι 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 $\mathrm{src}_n(E)$:

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

Fig 2(a) Illustrates the idea that for a definition inside a while loop, all the source nodes lie outside the loop. It shows a typical set of source nodes for the rhs of an assignment `X=E` at a program point p inside a loop.

Figure 2(b) shows a possible set of paths (obeying the existence constraints of the definition above) to the assignment `low=low+m` from its data source nodes. From this we can see that the rhs of the corresponding assignment inside the loop in the example from sect 3.1 has associated the set of source nodes {`high=in`, `b=15`, `low=0`}.

3.3 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

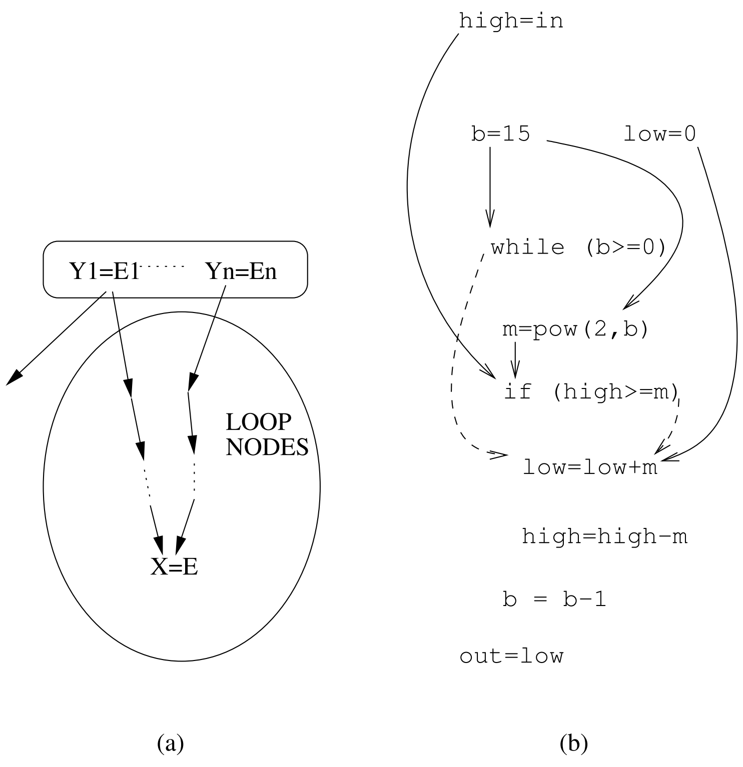

Fig. 2. Calculating source nodes within loops

potential problems with this assumption:

(i) 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;

(ii) 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. Note that we assume that the high inputs are *not* in the control of the attacker. Thus we are modelling a situation in which the environment delivering high inputs to the program is trusted, even though the program itself is not. This is appropriate for example in the analysis of untrusted code which is to be downloaded and run on a user's computer, where the user is the owner of the confidential data.

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 2.3) 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 $X^n_\lambda \stackrel{\text{def}}{=} X^n | L^\iota = \lambda$. 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 $\mathcal{L}^n_\lambda(X) \stackrel{\text{def}}{=} 0$ 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 $(\forall\lambda.\mathcal{L}^n_\lambda(X) \le a) \Rightarrow \mathcal{L}^n(X) \le a$

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 $Y \in H$.

3.4 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, $\mathcal{H}(\texttt{H}|\texttt{H} < 0)$ can take essentially any value between 0 and 31. (For the lower bound, let $p(h) = 0$ for the lowest $(2^{32} - 2^{16})$ values of h, $p(h) = 1/2^{16}$ for the rest. For the upper bound, let $b = 2^{31-a}$ and $ab - (1-b)\log(1-b) = 16$; then let $p(h) = 1/2^a$ when $h < 0$, $p(0) = 1 - b$, $p(h) = 0$ when $h > 0$.)

We deal with this difficulty by calculating bounds on the entropy of a hypothetical random variable $\hat{X}^n_\lambda$ which is defined everywhere. Let χ_n be the characteristic function of the domain of $\Delta(n)$: $\chi_n(\sigma) \stackrel{\text{def}}{=} 1$ if $\exists\sigma'.(\sigma,\sigma') \in \Delta(n)$; $\chi_n(\sigma) \stackrel{\text{def}}{=} 0$ otherwise. Then $\hat{X}^n_\lambda$ is defined to be a total random variable such that $X^n_\lambda = \hat{X}^n_\lambda | \chi_n = 1$. The $\hat{X}^n_\lambda$ are defined in the proof of correctness (theorem 3.2).

3.5 Analysis rules

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

(i) 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

(ii) 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 $\leq, \geq, =$) is to be read as asserting that $\mathcal{H}(\hat{E}^n_\lambda) \sim a$. 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 all remaining judgements, two things are implicit:

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

(ii) E is not a high variable 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, bits(E) is 1; if E is of Java's `int` type, 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 of information theory [3]: 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 example from sect 3.1. If we assume that `in` is uniformly distributed over $0 \leq i < 2^{16}$, it is easy to see, using the rules, that we get a leakage of all 16 bits into `low`.

3.6 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:*

(i) $E^n_\lambda = \hat{E}^n_\lambda | \chi_n = 1$

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

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

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

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

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

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

Table 1
Analysis rules

The proof of part 2 of the theorem hinges on the following lemma:

Lemma 3.4 *Let $\hat{E}^n_\lambda$ be as given by part 1 of the theorem. Then, if* $\text{src}_n(E) = \{n_1, \ldots, n_k\}$*, there exists a function f such that $\hat{E}^n_\lambda = f(\widehat{E(n_1)}^{n_1}_\lambda, \ldots, \widehat{E(n_k)}^{n_k}_\lambda)$.*

Correctness of the rules in table 1 follows because the entropy of $f(X)$ cannot be greater than that of X (the so called Data Processing Theorem of information theory [3]).

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 these rules may be refined to exploit the properties of the particular operators used in boolean and arithmetic expressions. The discovery of these refined rules is an on-going project. In the remainder of this section we present and justify improvements to the analysis of equality tests and some arithmetic operations. The resulting rules may be found in table 2. Note that these rules apply only *outside* loops.

4.1 Analysis of equality tests

In this section we develop a refined rule for analysis of tests of the form $E_1 == E_2$. The development reveals that calculating good *lower* bounds on the entropy of expressions at intermediate points will sometimes enable the calculation of good *upper* bounds on leakage. 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.

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 $\mathcal{H}$ (eqn. 1) gives:

$$\mathcal{U}_k(q) \overset{\text{def}}{=} q \log \frac{1}{q} + (1-q) \log \frac{2^k - 1}{1-q} \tag{3}$$

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 $\mathcal{H}(Y)$.

Proposition 4.1 *Given a k-bit random variable X and any other random variable Y, let $q \overset{\text{def}}{=} P(X = Y)$. Then $\mathcal{H}(X|Y) \leq \mathcal{U}_k(q)$.*

As an immediate corollary we have $\mathcal{H}(X) - \mathcal{H}(Y) \leq \mathcal{U}_k(q)$. Now the quantity of interest ($\mathcal{H}(X{==}Y)$) is just $\mathcal{B}(q)$, where

$$\mathcal{B}(q) \overset{\text{def}}{=} q \log \frac{1}{q} + (1-q) \log \frac{1}{1-q} \tag{4}$$

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 (that is, much less than 1) 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. 3. 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.)

As an example, consider the program P:

```
Y = H; [p]if (Y == 0) then [n0]X = 0 else [n1]X = 1 fi; [n2]Z = X
```

Suppose that $k = 32$ and the input distribution for H is uniform. 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 $\epsilon = \mathcal{B}(1/2^{32}) \approx 7.8 \times 10^{-9}$. Thus, using [Const] and [DP], we derive $n_2 \vdash [\texttt{Z}] \leq \epsilon$.

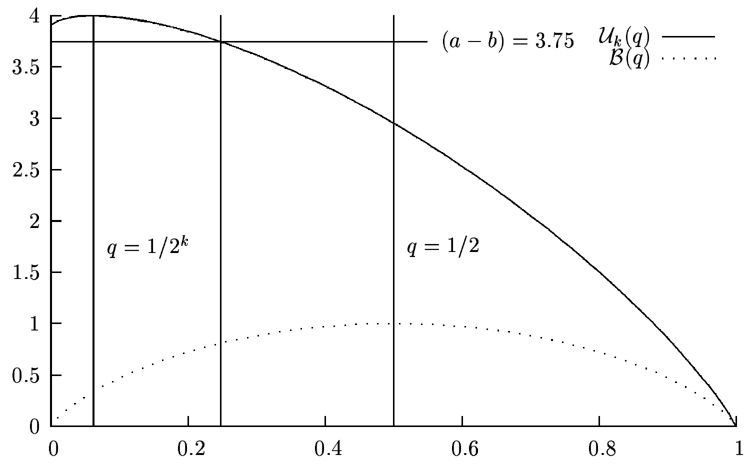

Fig. 3. the upper entropy for q in 4 bits

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

$$[\text{OpMax}]\ \frac{n \vdash [E_1] \leq b_1 \quad n \vdash [E_2] \leq b_2}{n \vdash [E_1 \odot E_2] \leq b_1 + b_2} \qquad [\text{Neg}]\ \frac{n \vdash [E] \sim a}{n \vdash [-E] \sim a}$$

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

$$[\text{ZeroMult}]\ \frac{}{n \vdash [E_1 * E_2] = 0}\ E_2 = 0$$

$$[\text{OddMult}]\ \frac{n \vdash [E_1] \sim a \quad n \vdash [E_2] = 0}{n \vdash [E_1 * E_2] \sim a}\ E_2 \text{ is odd}$$

Table 2
Some Refined Analysis rules (outside loops only)

4.2 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. We consider unary negation, addition and multiplication $(-, +, *)$ on the twos-complement representations of k-bit integers with overflow.

Unary negation is just a permutation and therefore easily seen to be an identity with respect to entropy, hence [Neg].

We use $\odot$ to mean any binary operator. For random variables X, Y, Z with $Z = X \odot Y$ we know (by the Data Processing Theorem) that $\mathcal{H}(Z) \leq \mathcal{H}(X) + \mathcal{H}(Y)$. This justifies [OpMax]. By itself, this rule gives no improvement over [DP]. However, we can achieve improved lower bounds (and hence, via [Eq], improved upper bounds) by exploiting the properties of $+$ and $*$.

As is well known, $+$ makes the set of k-bit numbers in twos-complement, a cyclic additive group with identity 0 and generator 1. The key group property which we exploit is that, whenever $x = y + z$, each of x, y, z is uniquely determined by the other two. This is sufficient to establish the following result, which in turn justifies [AddMin]:

Proposition 4.2 *Let* $Z \stackrel{\text{def}}{=} X + Y$*. Then*

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

Multiplication is less straightforward to analyse than addition as the algebraic structure of the operation is more complex. However, when one of the operands is a constant, we can sometimes establish good bounds on the entropy of the output. For example, $X * 0$ is always zero, thus $\mathcal{H}(X * 0) = 0$, hence [ZeroMult]. In fact, this can be seen as a consequence of a more general group property of $*$, namely that $\cdot * n$ generates a subgroup whose order is determined by the order of n.

In particular, the order of n is just that of the whole group (2^k) whenever n is odd. From this it follows that, for odd n, $\cdot * n$ (like unary negation) is an identity with respect to entropy. We note that, in this case, it is not necessary to establish the value of n, only that n is constant (once the low inputs are known) and that n is odd. Constancy is already captured by our analysis $(n \vdash [E] = 0)$ but a separate analysis would be required to establish parity. These facts are sufficient to justify [OddMult]. Establishing a more general formula, relating $\mathcal{H}(X)$, $\mathcal{H}(X * n)$ and the order of n, remains a subject for future work.

4.3 Correctness

Lemma 3.4, and hence theorem 3.2, extend easily to accommodate the additional rules in table 2. The key observation is that, in the cases where the new rules apply, we know not just that the function f of the lemma exists, but actually *which* function it is. The results above then justify the new rules directly.

5 Further 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, $\hat{X}^n_\lambda$ 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. The Data Processing theorem then implies that $\mathcal{H}(\hat{X}^n_\lambda) \leq \mathcal{H}(B) + \mathcal{H}(Y) + \mathcal{H}(Z)$. 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 the example of sect. 4.1 but the best we can derive for P' is the totally uninformative: $n \vdash [\texttt{Z}] \leq 32$. 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). This can 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 $\mathcal{H}(X|B=1) \leq k$ for any k-bit variable X, we are able to derive the much improved: $n \vdash [\texttt{Z}] \leq 32/2^{32} + \mathcal{B}(1/2^{32}) \approx 1.5 \times 10^{-8}$.

6 Conclusions and future work

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

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

References

[1] Clark, D., S. Hunt and P. Malacaria, *Quantitative analysis of the leakage of confidential data*, Electronic Notes in Theoretical Computer Science **59** (2002).

[2] Clark, D., S. Hunt and P. Malacaria, *Quantified interference for a while language*, Technical report, King's College London (2003).

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

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

[5] Goguen, J. and J. Meseguer, *Security policies and security models*, in: *IEEE Symposium on Security and Privacy* (1982), pp. 11–20.

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

[7] McIver, A. and C. Morgan, *A probabilistic approach to information hiding*, in: *Programming methodology*, Springer-Verlag New York, Inc., 2003 pp. 441–460.

[8] Millen, J., *Covert channel capacity*, in: *Proc. 1987 IEEE Symposium on Research in Security and Privacy* (1987).

[9] Pierro, A. D., C. Hankin and H. Wiklicky, *Approximate non-interference*, in: I. Cervesato, editor, *CSFW'02 – 15th IEEE Computer Security Foundation Workshop* (2002).

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

[11] Sabelfeld, A. and D. Sands, *A per model of secure information flow in sequential programs*, in: *Proc. European Symposium on Programming* (1999).

[12] Sabelfeld, A. and D. Sands, *Probabilistic noninterference for multi-threaded programs*, in: *Proc. 13th IEEE Computer Security Foundations Workshop* (2000).

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

[14] W. Gray, J., III, *Toward a mathematical foundation for information flow security*, in: *Proc. 1991 IEEE Symposium on Security and Privacy*, Oakland, CA, 1991, pp. 21–34.