# Program Analysis Probably Counts

Alessandra Di Pierro[1], Chris Hankin[2], and Herbert Wiklicky[2]

[1] University of Verona, Ca' Vignal 2 - Strada le Grazie 15 I-37134 Verona, Italy
[2] Imperial College London, 180 Queen's Gate London SW7 2AZ, UK

**Abstract.** Semantics-based program analysis uses an abstract semantics of programs/systems to statically determine run-time properties. Classic examples from compiler technology include analyses to support constant propagation and constant folding transformations and estimation of pointer values to prevent buffer overruns. More recent examples include the estimation of information flows (to enforce security constraints) and estimation of non-functional properties such as timing (to determine worst case execution times in hard real-time applications).
The classical approaches are based on semantics involving discrete mathematics. Paralleling trends in model-checking, there have been recent moves towards using probabilistic and quantitative methods in program analysis.
In this paper we will start by reviewing both classical and probabilistic/quantitative approaches to program analysis. We will provide a comparison of the two approaches. We will use a simple information flow analysis to exemplify the classical approach. The existence of covert information flows through timing channels are difficult to detect using classical techniques; we show how such problems can be addressed using probabilistic techniques.

## 1 Introduction

Program analysis is a collection of techniques to predict in advance what will happen when a program is executed. Classically, such information could be used to optimise the code produced by a compiler; more recently this has formed the basis for the automatic debugging, verification and certification of code. Unfortunately, well known fundamental results, like the Halting problem, tell us that it is impossible to know everything about the behaviour of every program. The solution to this obstacle of undecidabilty is to aim for partial answers to some of the questions. Program analysis techniques have been in use in compilers since the dawn of the computing age; semantics-based techniques, which will be our focus, stem from the late 1970s.

Different applications and users have different priorities and interests and therefore accept different kinds of imprecision. When it comes, for example, to systems which are critical for life and limb one might be cautious and attempt to determine absolute limits on what can go wrong in the worst case – like in the case of safety critical systems in cars, planes, etc. If, on the other hand, the possible damage is only in terms of lost money, time or other resource one might

be inclined to accept an estimate in order to forecast average profits or losses – as in the context of speculative threading, power consumption of mobile devices, etc.

In this paper we will briefly compare and contrast two different approaches to semantics-based program analysis. The first is based on abstract interpretation and uses traditional (discrete) mathematical models as a basis. The alternative is based on probabilistic and continuous models – making program analysis count (probably)! We will use language-based security as an example of what can be achieved by traditional methods and then show how quantitative methods can help detect and fix indirect information flows known as covert channels.

## 2 Program Analysis

Abstract Interpretation [1–4] provides a general methodology for constructing static analyses which is, to some extent, independent of the particular style used to specify the program analysis. Thus, it applies to any formulation of a (data/control flow or type/effect-system) analysis.

The *semantics* of a program $f$ identifies some set $V$ of values and specifies how the program transforms one value $v_1$ to another $v_2$: $f \vdash v_1 \rightsquigarrow v_2$. In a similar way, a *program analysis* identifies the set $L$ of properties and specifies how a program $f$ transforms one property $l_1$ to another $l_2$: $f \vdash l_1 \rhd l_2$. For first-order program analyses, i.e. those that abstract properties of values, correctness is established by directly relating properties to values using a *correctness relation*: $R : V \times L \to \{\text{TRUE}, \text{FALSE}\}$. The intention is that $v\ R\ l$ formalises our claim that the value $v$ is described by the property $l$.

To be useful one has to prove that the correctness relation $R$ is preserved under computation: if the relation holds between the initial value and the initial property then it also holds between the final value and the final property:

$$v_1\ R\ l_1\ \ \wedge\ \ f \vdash v_1 \rightsquigarrow v_2\ \ \wedge\ \ f \vdash l_1 \rhd l_2 \Rightarrow v_2\ R\ l_2$$

The most common scenario in abstract interpretation is when both $V$ and $L$ are complete lattices. We then require that $R$ satisfies:

$$v\ R\ l_1\ \ \wedge\ \ l_1 \sqsubseteq l_2 \Rightarrow v\ R\ l_2 \tag{1}$$

$$(\forall l \in L' \subseteq L : v\ R\ l) \Rightarrow v\ R\ \left(\bigsqcap L'\right) \tag{2}$$

The first of these concerns *safety* [4]: if we have a property which correctly describes a value, then any larger property is also a safe description. The second concerns the existence of *best* descriptions: if we have a set of properties that correctly describe a value then their meet will also be a correct description and is more accurate.

The correctness relation is often achieved via a *Galois connection*: $(V, \alpha, \gamma, L)$ is a Galois connection between the complete lattices $(V, \sqsubseteq)$ and $(L, \sqsubseteq)$ if and only

if $\alpha : V \to L$ and $\gamma : L \to V$ are monotone functions that satisfy: $\gamma \circ \alpha \sqsupseteq id_V$ and $\alpha \circ \gamma \sqsubseteq id_L$.

Having defined a suitable "set" of properties we then define suitable interpretations of program operations. The framework of abstract interpretation guarantees that the analysis will be safe as long as we use an interpretation, $F_{\text{abs}}$, of each language operator, $F$, that satisfies: $F_{\text{abs}} \sqsupseteq \alpha \circ F \circ \gamma$.

Quantitative approaches to program analysis aim at developing techniques which provide approximate answers (in a way similar to the classical program analysis) together with some numerical estimate of the approximation introduced by the analysis.

One useful source of numerical information for a quantitative program analysis is a probabilistic semantics and in particular the use of vector space or linear algebraic structures for modelling the computational domain. By exploiting the probabilistic information resulting from a probabilistic program analysis one can quantify the precision of the analysis and obtain as a result answers which are for example "approximate up to 35%".

As a quantitative approach to program analysis we have developed Probabilistic Abstract Interpretation (PAI) [5, 6] which recasts classical Abstract Interpretation in a probabilistic setting where linear spaces replace the classical order-theoretic domains, and the notion of the so-called *Moore-Penrose pseudo-inverse* of a linear operator replaces the classical notion of a Galois connection. The abstractions we get this way are *close* approximations of the concrete semantics. Thus, closeness is a quantitative replacement for classical safety which does not require any approximation ordering.

The application of operator algebraic methods instead of order theoretic ones makes the framework of probabilistic abstract interpretation essentially different from approaches which apply classical abstract interpretation to probabilistic domains [7]. Although classical techniques can also be used in a probabilistic context, e.g. to approximate distributions, as was demonstrated for example in a number of papers by D.Monniaux [7], this will always result in safe, i.e. worst case analysis.

Probabilistic Abstract Interpretation is defined in general for infinite dimensional Hilbert spaces. We recall here the general definition, although in this paper we will only consider the finite dimensional case. Given two probabilistic domains $\mathcal{C}$ and $\mathcal{D}$, a *probabilistic abstract interpretation* is defined by a pair of linear maps, $\mathbf{A} : \mathcal{C} \mapsto \mathcal{D}$ and $\mathbf{G} : \mathcal{D} \mapsto \mathcal{C}$, between the concrete domain $\mathcal{C}$ and the abstract domain $\mathcal{D}$, such that $\mathbf{G}$ is the *Moore-Penrose pseudo-inverse* of $\mathbf{A}$, and vice versa. Let $\mathcal{C}$ and $\mathcal{D}$ be two Hilbert spaces and $\mathbf{A} : \mathcal{C} \mapsto \mathcal{D}$ a bounded linear map between them. A bounded linear map $\mathbf{A}^\dagger = \mathbf{G} : \mathcal{D} \mapsto \mathcal{C}$ is the Moore-Penrose pseudo-inverse of $\mathbf{A}$ iff

$$\mathbf{A} \circ \mathbf{G} = \mathbf{P}_A \quad \text{and} \quad \mathbf{G} \circ \mathbf{A} = \mathbf{P}_G$$

where $\mathbf{P}_A$ and $\mathbf{P}_G$ denote orthogonal projections (i.e. $\mathbf{P}_A^* = \mathbf{P}_A = \mathbf{P}_A^2$ and $\mathbf{P}_G^* = \mathbf{P}_G = \mathbf{P}_G^2$) onto the ranges of $\mathbf{A}$ and $\mathbf{G}$.

Alternatively, if $\mathbf{A}$ is Moore-Penrose invertible, its Moore-Penrose pseudo-inverse, $\mathbf{A}^\dagger$ satisfies the following:

$$\textbf{(i)}\ \mathbf{A}\mathbf{A}^\dagger\mathbf{A} = \mathbf{A}, \qquad \textbf{(iii)}\ (\mathbf{A}\mathbf{A}^\dagger)^* = \mathbf{A}\mathbf{A}^\dagger,$$
$$\textbf{(ii)}\ \mathbf{A}^\dagger\mathbf{A}\mathbf{A}^\dagger = \mathbf{A}^\dagger, \qquad \textbf{(iv)}\ (\mathbf{A}^\dagger\mathbf{A})^* = \mathbf{A}^\dagger\mathbf{A}.$$

where $\mathbf{M}^*$ is the adjoint of $\mathbf{M}$. It is instructive to compare these equations with Galois connections where we similarly have $\alpha \circ \gamma \circ \alpha = \alpha$ and $\gamma \circ \alpha \circ \gamma = \gamma$.

As in the classical framework, given a concrete semantics we can always construct a *best correct approximation* for this semantics, although the notions of correctness and optimality assume a different connotation in our linear setting. If $\Phi$ is a linear operator on some vector space $\mathcal{V}$ expressing the probabilistic semantics of a concrete system, and $\mathbf{A} : \mathcal{V} \mapsto \mathcal{W}$ is a linear abstraction function from the concrete domain into an abstract domain $\mathcal{W}$, we can compute the (unique) Moore-Penrose pseudo-inverse $\mathbf{G} = \mathbf{A}^\dagger$ of $\mathbf{A}$. An abstract semantics can then be defined as the linear operator on the abstract domain $\mathcal{W}$:

$$\Psi = \mathbf{A} \circ \Phi \circ \mathbf{G} = \mathbf{G}\Phi\mathbf{A}.$$

## 3 Approximations: Classical Function Interpolation

In oder to give a deeper comparison of these two approaches, we need a setting where the domain in question in some way naturally provides both structures. One such situation is in the context of classical function interpolation or approximation.

The set of real-valued functions on real interval $[a, b]$ obviously comes with a canonical partial order, namely the point-wise ordering, and at the same time is equipped with a vector space structure, again the point-wise addition and scalar multiplication. Some care has to be taken in order to define an inner product, e.g. one could consider only the square integrable function $L^2([a, b])$. In order to avoid mathematical (e.g. measure-theoretic) details we simplify the situation by just considering the step functions on the interval.

For a (closed) real interval $[a, b] \subseteq \mathbb{R}$ we call the set of subintervals $[a_i, b_i]$ with $i = 1, \ldots, n$ an *n-subdivision* of $[a, b]$ if $\bigcup_{i=1}^{n}[a_i, b_i] = [a, b]$ and $b_i - a_i = \frac{b-a}{n}$ for all $i = 1, \ldots, n$. We assume that the subintervals are enumerated in the obvious way, i.e. $a_i < b_i = a_{i+1} < b_{i+1}$ for all $i$ and in particular that $a = a_1$ and $b_n = b$.

**Definition 1.** The set of *n-step functions* $\mathcal{T}_n([a, b])$ on $[a, b]$ is the set of real-valued functions $f : [a, b] \to \mathbb{R}$ such that $f$ is constant on each subinterval $(a_i, b_i)$ in the n-subdivision of $[a, b]$.

We define a partial order on $\mathcal{T}_n([a, b])$ in the obvious way: for $f, g \in \mathcal{T}_n([a, b])$:

$$f \sqsubseteq g \text{ iff } f(\frac{b_i - a_i}{2}) \leq g(\frac{b_i - a_i}{2})$$

i.e. iff the value of $f$ (which we obtain by evaluating it on the mid-point in $(a_i, b_i)$) on all subintervals $(a_i, b_i)$ is less or equal to the value of $g$.

It is also obvious to see that $\mathcal{T}_n([a, b])$ has a vector space structure isomorphic to $\mathbb{R}^n$ and thus is also provided with an inner product. More concretely we

define the vector space operations $. \cdot . : \mathbb{R} \times \mathcal{T}_n([a,b]) \to \mathcal{T}_n([a,b])$ and $. + . : \mathcal{T}_n([a,b]) \times \mathcal{T}_n([a,b]) \to \mathcal{T}_n([a,b])$ pointwise as follows:

$$(\alpha \cdot f)(x) = \alpha f(x)$$

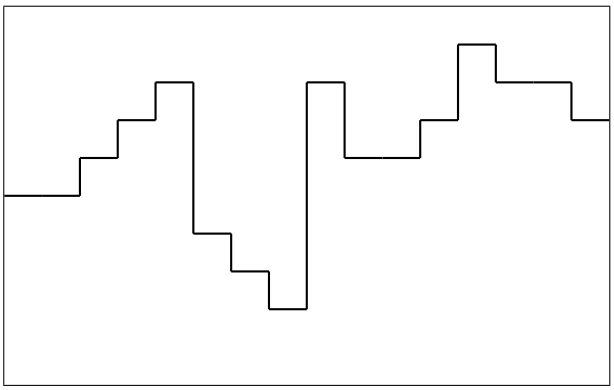
$$(f + g)(x) = f(x) + g(x)$$

for all $\alpha \in \mathbb{R}$, $f, g \in \mathcal{T}_n([a,b])$ and $x \in [a,b]$. The inner product is given by:

$$\langle f, g \rangle = \sum_{i=1}^{n} f(\frac{b_i - a_i}{2}) g(\frac{b_i - a_i}{2}).$$

In this setting we now can apply and compare both the classical and the quantitative version of abstract interpretation as in the following example.

*Example 2.* Let us consider a step function $f$ in $\mathcal{T}_{16}$ (the concrete values of $a$ and $b$ don't really play a role in our setting) which can be depicted as:



which can also be represented by a vector in $\mathbb{R}^{16}$:

$$\big(5\ 5\ 6\ 7\ 8\ 4\ 3\ 2\ 8\ 6\ 6\ 7\ 9\ 8\ 8\ 7\big)$$

We then can construct a series of abstractions which correspond to coarser and coarser sub-divisions of the interval $[a,b]$, e.g. considering 8, 4 etc. subintervals instead of the original 16. These abstractions are from $\mathcal{T}_{16}([a,b])$ to $\mathcal{T}_8([a,b])$, $\mathcal{T}_4([a,b])$ etc. and can be represented by $16 \times 8$, $16 \times 4$, etc. matrices.

For example, the abstraction which joins two sub-intervals and which corresponds to the abstraction $\alpha_8 : \mathcal{T}_{16}([a,b]) \to \mathcal{T}_8([a,b])$ together with its Moore-

Penrose pseudo-inverse is represented by:

$$\mathbf{A}_8 = \begin{pmatrix} 1&0&0&0&0&0&0&0\\ 1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1\\ 0&0&0&0&0&0&0&1 \end{pmatrix} \quad \mathbf{G}_8 = \begin{pmatrix} \frac{1}{2}&\frac{1}{2}&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&\frac{1}{2}&\frac{1}{2}&0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&\frac{1}{2}&\frac{1}{2}&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&\frac{1}{2}&\frac{1}{2}&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&\frac{1}{2}&\frac{1}{2}&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&\frac{1}{2}&\frac{1}{2}&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0&\frac{1}{2}&\frac{1}{2}&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&\frac{1}{2}&\frac{1}{2} \end{pmatrix}$$

With the help of $\mathbf{A}_i$ we can easily compute the the abstraction of $f$ as $f\mathbf{A}$, which in order to compare it with the original $f$ we can then again concretise using $\mathbf{G}$, i.e. computing $f\mathbf{AG}$. In a similar way we can also compute the over- and under-approximation of $f$ in $\mathcal{T}_i$ based on the above pointwise ordering and its reverse ordering. The result of these abstractions is depicted geometrically in Figure 1.

The individual diagrams in this figure depict the original, i.e. concrete step function $f \in \mathcal{T}_{16}$ together with its approximations in $\mathcal{T}_8$, $\mathcal{T}_4$, etc. On the left hand side the PAI abstractions show how coarser and coarser interval subdivisions result in a series of approximations which try to interpolate the given function as closely as possible, sometimes below, sometimes above the concrete values. The diagrams on the right hand side depict the classical over- and under-approximations: In each case the function $f$ is entirely below or above these approximations, i.e. we have safe but not necessarily close approximations. Additionally, one can also see from these figures not only that the PAI interpolation is in general closer to the original function than the classical abstractions (in fact it is the closest possible) but also that the PAI interpolation is always between the classical over- and under-approximations.

## 4 Information Flow

We now turn to an example which is closer to programming language theory and is motivated by language-based security concerns. We focus on the flow of information between variables in a program. Classical work in language-based security partitions the variables into different security classes and then seeks to enforce various security properties such as *no read-up* (where, for example,

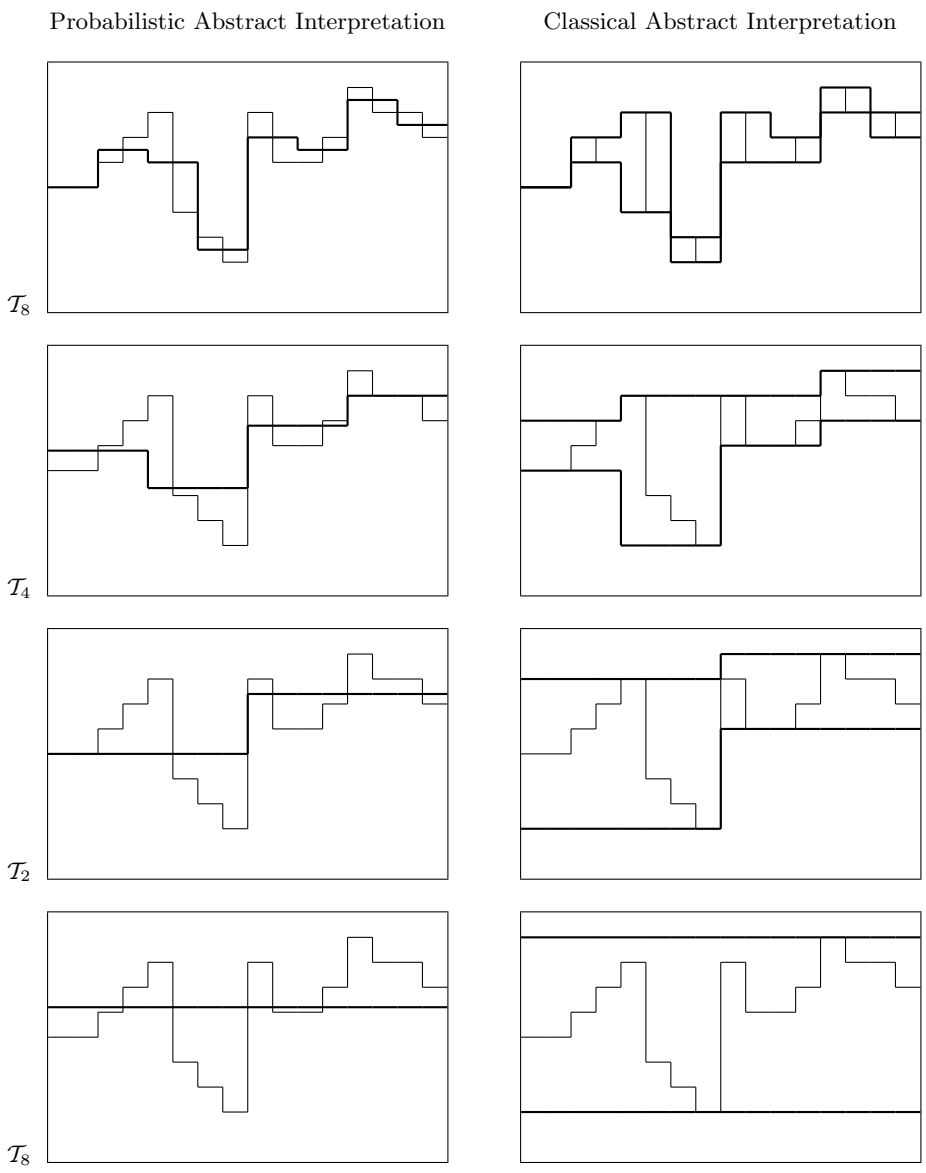



**Fig. 1.** Average, Over- and Under-Approximation

the value of a higher security variable may not be assigned to a lower security variable). The material in this section is based on [8].

Following Denning [9] we divide information flows into two classes: *direct* and *indirect*. Indirect flows are just the transitive flows (a flow from $x$ to $y$ followed by a flow from $y$ to $z$ implies a flow from $x$ to $z$). The direct flows are further divided:

1. *Explicit* flows arise from assignments; for example, $x := y + z$ causes explicit information flows from both $y$ and $z$ to $x$.
2. *Implicit* flows arise when one variable's value influences the value assigned to another by determining the flow of control. There are two types of implicit flow (though Denning only considered the first type in detail):
   (a) *Local flows* arise from guards in conditionals:

   $$\texttt{if} \quad x \quad \texttt{then} \quad y := z \quad \texttt{else} \quad y := w.$$

   Here there is local implicit information flow from $x$ to $y$, in addition to the explicit flows from $z$ and $w$.
   (b) *Global flows* arise from guards in while loops

   $$x := y; (\texttt{while} \quad w \quad \texttt{do} \quad x := z); \cdots.$$

   Here there is a global implicit flow from $w$ to all subsequent program points, since reaching those points carries the information that the loop terminated and hence, in this example, that $w$ was **false**.

### 4.1 Information Flow for While

The syntax of the language is as follows:

$$\begin{gathered} S \in \textbf{Statement} \\ C \in \textbf{Command} \\ \ell \in \textbf{Lab} \\ x \in \textbf{Ide} \\ a \in \textbf{Arith-exp} \\ b \in \textbf{Bool-exp} \end{gathered}$$

$$\begin{aligned} S &::= C^{\ell} \\ C &::= \texttt{skip} \mid x := a \mid S_1; S_2 \mid \\ &\quad\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ b\ \texttt{do}\ S \mid \\ &\quad\ \texttt{new}\ x.\ S \end{aligned}$$

This is the standard While language, used in many programming language textbooks, extended with a block structuring construct (`new`); the reader should see [4] for a detailed treatment of program analysis techniques for this language. Command are labelled in order to allow us to formulate the program analysis; the labels play no part in the semantics.

The semantics of the language is standard. The Information Flow Analysis is presented in the style of a *flow logic* as pioneered by Flemming Nielson and Hanne Riis Nielson and their group; see [4] for an introduction to this style. For this language the control flow is explicit and we just use the flow logic to specify information flow. We assume that **Ide** and **Lab** are restricted to those variables and labels appearing in the program, ensuring that the constraints have finite solutions. We write

$$(\widehat{\mathsf{G}}, \widehat{\mathsf{D}}) \models S$$

when $(\widehat{\mathsf{G}}, \widehat{\mathsf{D}})$ is an acceptable Information Flow Analysis of the statement $S$.

We use the following functions on labels:

$$\begin{aligned}
\widehat{\mathsf{X}} \in \mathbf{Assign} &= \mathbf{Lab} \to \mathcal{P}(\mathbf{Ide}) \\
\widehat{\mathsf{G}} \in \mathbf{Global} &= \mathbf{Lab} \to \mathcal{P}(\widehat{\mathbf{Ide}}) \\
\widehat{\mathsf{D}} \in \mathbf{Dep} \quad &= \mathbf{Lab} \to \mathcal{P}(\widehat{\mathbf{Ide}} \times \widehat{\mathbf{Ide}})
\end{aligned}$$

where $\widehat{\mathbf{Ide}} = \mathbf{Ide} \cup \{\bullet\}$. Given a label, $\widehat{\mathsf{X}}$ returns a superset of those variables which may be assigned to during evaluation of the statement with that label. It is necessary to collect this information in order to handle implicit flows correctly. The function $\widehat{\mathsf{X}}$ has a straightforward inductive definition (see [8].

Given a label, $\widehat{\mathsf{G}}$ returns a superset of those variables whose values may affect termination of the statement with that label. The special variable $\bullet$ is used to indicate that a while loop has been encountered (and thus non-termination is possible). The variables in $\widehat{\mathsf{G}}(\ell)$ are involved in implicit global flows to any variables assigned to after the execution of the statement at $\ell$.

Given a label, $\widehat{\mathsf{D}}$ returns a superset of the dependencies for the statement with that label; a pair $(x, y)$ is a dependency for $S$ if different values for $y$ in the state prior to execution of $S$ can result in different values for $x$ in the state after execution of $S$.

We also use $\mathrm{FV}(a)$ to denote the set of variables occurring in $a$ (and similarly for $b$). Note that $\widehat{\mathsf{D}}(\ell)$ is a binary relation on $\widehat{\mathbf{Ide}}$. We denote the identity relation on $\widehat{\mathbf{Ide}}$ by Id. We use $;$ for relational composition and also overload this notation to allow the 'composition' of a set with a relation, thus: $Y ; R \stackrel{\text{def}}{=} \{z \mid \exists y \in Y.\ y\ R\ z\}$. Where convenient, we treat $\widehat{\mathsf{D}}(\ell)$ as a function of type $\widehat{\mathbf{Ide}} \to \mathcal{P}(\widehat{\mathbf{Ide}})$. In particular, we use a 'function update' notation on relations thus: $R[x \mapsto Y] \stackrel{\text{def}}{=} \{(y, z) \in R \mid y \neq x\} \cup \{x\} \times Y$.

A selection of rules from the Information Flow Analysis are shown in table 1.

If a variable is not assigned at a label, its post-execution value will be the same as its pre-execution value; hence the clause for `skip`. When an assignment is executed, there are direct information flows from the variables on the right hand side to the variable on the left hand side. In a sequential composition, the variables that might affect termination are those that might affect the termination of the first sub-statement or those that might affect termination of the second sub-statement after taking account of dependencies created by the first

$$(\widehat{\mathsf{G}},\widehat{\mathsf{D}}) \models \texttt{skip}^{\ell} \text{ iff } \widehat{\mathsf{D}}(\ell) \supseteq \mathrm{Id}$$

$$\begin{array}{l}(\widehat{\mathsf{G}},\widehat{\mathsf{D}}) \models (x := a)^{\ell} \\ \quad \text{iff } \widehat{\mathsf{D}}(\ell) \supseteq \mathrm{Id}[x \mapsto \mathrm{FV}(a)]\end{array}$$

$$\begin{array}{l}(\widehat{\mathsf{G}},\widehat{\mathsf{D}}) \models (C_1^{\ell_1}; C_2^{\ell_2})^{\ell} \\ \quad \text{iff } (\widehat{\mathsf{G}},\widehat{\mathsf{D}}) \models C_1^{\ell_1} \wedge (\widehat{\mathsf{G}},\widehat{\mathsf{D}}) \models C_2^{\ell_2} \wedge \\ \quad\quad \widehat{\mathsf{G}}(\ell) \supseteq \widehat{\mathsf{G}}(\ell_1) \cup \widehat{\mathsf{G}}(\ell_2)\,;\widehat{\mathsf{D}}(\ell_1) \wedge \\ \quad\quad \widehat{\mathsf{D}}(\ell) \supseteq \widehat{\mathsf{D}}(\ell_2)\,;\widehat{\mathsf{D}}(\ell_1)\end{array}$$

$$\begin{array}{l}(\widehat{\mathsf{G}},\widehat{\mathsf{D}}) \models (\texttt{if } b \texttt{ then } C_1^{\ell_1} \texttt{ else } C_2^{\ell_2})^{\ell} \\ \quad \text{iff } (\widehat{\mathsf{G}},\widehat{\mathsf{D}}) \models C_1^{\ell_1} \wedge (\widehat{\mathsf{G}},\widehat{\mathsf{D}}) \models C_2^{\ell_2} \wedge \\ \quad\quad \widehat{\mathsf{G}}(\ell) \supseteq \widehat{\mathsf{G}}(\ell_1) \cup \widehat{\mathsf{G}}(\ell_2) \wedge \\ \quad\quad (\bullet \in \widehat{\mathsf{G}}(\ell) \Rightarrow \widehat{\mathsf{G}}(\ell) \supseteq \mathrm{FV}(b)) \wedge \\ \quad\quad \widehat{\mathsf{D}}(\ell) \supseteq \widehat{\mathsf{D}}(\ell_1) \cup \widehat{\mathsf{D}}(\ell_2) \wedge \\ \quad\quad \widehat{\mathsf{D}}(\ell) \supseteq \widehat{\mathsf{X}}(\ell) \times \mathrm{FV}(b)\end{array}$$

**Table 1.** Information Flow Analysis for While

sub-statement (hence the relational composition); the dependencies are generated by the relational composition of the dependencies for the sub-statements. The conditional clause mainly concerns propagation of information from the then and else branches; if either branch contains a while loop, the variables appearing in the predicate are added to $\widehat{\mathsf{G}}(\ell)$ – this ensures that there is an implicit global flow to statements following the conditional; the variables assigned in the branches are forced to depend on the variables in the predicate – this establishes the implicit local information flows.

Having analysed a program, $C^{\ell}$, we determine that there is no breach of security if both

- $\{x \mid x \in H\} \cap \widehat{\mathsf{G}}(\ell) = \emptyset$, and
- $\forall x \in L.\ \nexists y \in H . x\ \widehat{\mathsf{D}}(\ell)\ y$

i.e. there are no global information flows from high variables and no low variable depends on any high variable.

## 5 Adding Quantities

Early work on language-based security, such as Volpano and Smith's type systems [10], precluded the use of high security variables to affect control flow. Specifically, the conditions in if-commands and while-commands were restricted to using only low security information. If this restriction is weakened, as we have done in the previous section, it opens up the possibility that high security

data may be leaked through the different timing behaviour of alternative control paths. This kind of leakage of information is said to form a *covert timing channel* and is a serious threat to the security of programs (cf. e.g. [11]).

We consider a language similar to that used in the previous section but with the addition of a probabilistic choice construct and skip statements of different duration (reflecting the fact that assignment consumes less cycles than a conditional). We draw motivation from [12]:

$$C, D ::= x := a \mid \textbf{skipAsn}\ x\ a \mid \textbf{if}\ b\ \textbf{then}\ C\ \textbf{else}\ D \mid \textbf{skipIf}\ b\ C \mid \textbf{while}\ b\ \textbf{do}\ C \mid C; D \mid [\textbf{choose}]^p\ C\ \textbf{or}\ D$$

The probabilistic choice is used in an essential way in the program transformation presented later. We also keep the language of types from [12], although in a simplified form (with $\mathrm{L} \leq \mathrm{H}$ and $s \leq s$):

| | | |
|---|---|---|
| Security levels | $s ::= \mathrm{L} \mid \mathrm{H}$ | |
| Base types | $\overline{\tau} ::= \textbf{Int} \mid \textbf{Bool}$ | and sub-typing: $\dfrac{s_1 \leq s_2}{\overline{\tau}_{s_1} \leq \overline{\tau}_{s_2}}$. |
| Security types | $\tau ::= \overline{\tau}_s$ | |

We will indicate by $E$ the state of a computation and denote by $E_L$ its restriction to low variables, i.e. a state which is defined as $E$ for all the low variables for which $E$ is defined, and is undefined otherwise. We say that two configurations $\langle E \mid C\rangle$ and $\langle E' \mid C'\rangle$ are *low equivalent* if and only if $E_L = E'_L$ and we indicate this by $\langle E \mid C\rangle =_L \langle E' \mid C'\rangle$. In the following we will sometimes use the shorthand notation $c, c_1, c_2, \ldots, c', c'_1, \ldots$ for configurations. We will also denote by **Conf** the set of all configurations.

The big step semantics of expressions and the small-step semantics of commands are essentially the same as those in [12]. The only difference is the rule for probabilistic choice which we have added to the original semantics (see Table 2). In this rule, $t_{ch}$ indicates the time it takes to execute a choice command. In general, we will use the time labels $t_{\cdot}$ to represent the time it takes to perform certain operations: $t_x$ is the time to store a variable, $t_a$ is the time it takes to evaluate an arithmetic expression, $t_{asn}$ represents the time to perform an assignment, $t_{br}$ is the time required for a branching step, and $t_{ch}$ is the time to perform a probabilistic choice. The choice rule states that the execution of a probabilistic choice construct leads, after a time $t_{ch}$, to a state where either the command $C$ or the command $D$ is executed with probability $p$ or $1 - p$, respectively.

### 5.1 Abstract Semantics

According to the notion of security we consider in this paper, an observer or attacker can only observe the changes in low variables. Therefore, we can simplify the semantics by 'collapsing' the execution tree in such a way that execution steps during which the value of all low variables is unchanged are combined into one single step. We call an execution sequence $\sigma$ *deterministic* if the probability of the sequence is 1 and we call it *low stable* if all the states in the low-slice are equal in

(Assign) $$\frac{E \vdash a \Downarrow v}{\langle E \mid x := a\rangle \xrightarrow{1:t_a \cdot t_x \cdot t_{asn} \cdot \surd} E[x = v]}$$

(Seq) $$\frac{\langle E \mid C\rangle \xrightarrow{p:ts \cdot \surd)} E'}{\langle E \mid C; D\rangle \xrightarrow{p:ts} \langle E' \mid D\rangle}$$

$$\frac{\langle E \mid C\rangle \xrightarrow{p:ts} \langle E' \mid C'\rangle}{\langle E \mid C; D\rangle \xrightarrow{p:ts} \langle E' \mid C'; D\rangle}$$

(If) $$\frac{E \vdash b \Downarrow \text{TRUE}}{\langle E \mid \textbf{if } b \textbf{ then } C \textbf{ else } D\rangle \xrightarrow{1:t_b \cdot t_{br}} \langle E \mid C\rangle}$$

$$\frac{E \vdash b \Downarrow \text{FALSE}}{\langle E \mid \textbf{if } b \textbf{ then } C \textbf{ else } D\rangle \xrightarrow{1:t_b \cdot t_{br}} \langle E \mid D\rangle}$$

(SkipAsn) $$\frac{E \vdash a \Downarrow v}{\langle E \mid \textbf{skipAsn } x\ a\rangle \xrightarrow{1:t_a \cdot t_{asn} \cdot \surd} E}$$

(SkipIf) $$\frac{E \vdash b \Downarrow v}{\langle E \mid \textbf{skipIf } b\ C\rangle \xrightarrow{1:t_b \cdot t_{br}} \langle E \mid C\rangle} \quad v \in \{\text{TRUE}, \text{FALSE}\}$$

(While) $$\frac{E \vdash b \Downarrow \text{FALSE}}{\langle E \mid \textbf{while } b \textbf{ do } C\rangle \xrightarrow{1:t_b \cdot t_{br} \cdot \surd} E}$$

$$\frac{E \vdash b \Downarrow \text{TRUE}}{\langle E \mid \textbf{while } b \textbf{ do } C\rangle \xrightarrow{1:t_b \cdot t_{br}} \langle E \mid C; \textbf{while } b \textbf{ do } C\rangle}$$

(Choose) $$\frac{}{\langle E \mid [\textbf{choose}]^p\ C \textbf{ or } D\rangle \xrightarrow{p:t_{ch}} \langle E \mid C\rangle}$$

$$\frac{}{\langle E \mid [\textbf{choose}]^p\ C \textbf{ or } D\rangle \xrightarrow{(1-p):t_{ch}} \langle E \mid D\rangle}$$

**Table 2.** pWhile semantics

the low variables. The empty path (of length zero) is by definition deterministic and low stable. An execution sequence is *maximal deterministic/low stable* if it is not a proper sub-sequence of another deterministic/low stable path.

**Definition 3.** The collapsed transition relation $\langle E_1 \mid C_1\rangle \stackrel{p:T}{\Longrightarrow} \langle E_2 \mid C_2\rangle$ between two configurations is defined iff

**(i)** there exists a configuration $\langle E_1' \mid C_1'\rangle$ such that $\langle E_1 \mid C_1\rangle \xrightarrow{p:t} \langle E_1' \mid C_1'\rangle$,
**(ii)** $\langle E_1' \mid C_1'\rangle \xrightarrow{1:t_1} \ldots \langle E_2' \mid C_2'\rangle \xrightarrow{1:t_n} \langle E_2 \mid C_2\rangle$ is deterministic,
**(iii)** $\langle E_1 \mid C_1\rangle \xrightarrow{p:t} \langle E_1' \mid C_1'\rangle \ldots \xrightarrow{1:t_{n-1}} \langle E_2' \mid C_2'\rangle$ is maximal low stable,
**(iv)** and $T = t + \sum_{i=1}^{n} t_i$.

### 5.2 Bisimulation and Timing Leaks

Observing the low variables and the running time separately is not the same as observing them together; a correlation between the two random variables (probability and time) has to be taken into account. A naive probabilistic extension of the $\Gamma$-bisimulation notion introduced in [12] might not take this into account. More precisely, this may happen if time and probability are treated as two independent aspects which are observed separately in a mutual exclusive way. According to such a notion an attacker must set up two different covert channels if he/she wants to exploit possible interference through both the probabilistic and the timing behaviour of the system. The notion of bisimulation we introduce here allows us to define a stronger security condition: an attacker must be able to distinguish the probabilities that two programs compute a given result in a given execution time. This is obviously different from being able to distinguish the probability distributions of the results *and* the running time.

Probabilistic bisimulation was first introduced in [13] and refers to an equivalence on probability distributions over the states of the processes. This latter equivalence is defined as a lifting of the bisimulation relation on the support sets of the distributions, namely the states themselves.

An equivalence relation $\sim \subseteq S \times S$ on $S$ can be lifted to a relation $\sim^* \subseteq \mathbf{Dist}(S) \times \mathbf{Dist}(S)$ between probability distributions on $S$ via (cf [14, Thm 1]): $\mu \sim^* \nu$ iff $\forall [s] \in S/_\sim : \mu([s]) = \nu([s])$. It follows that $\sim^*$ is also an equivalence relation ([14, Thm 3]). For any equivalence relation $\sim$ on the set **Conf** of configurations, we define the associated *low equivalence* relation $\sim_L$ by $c_1 \sim_L c_2$ if $c_1 \sim c_2$ and $c_1 =_L c_2$. Obviously $\sim_L$ is again an equivalence relation. We can lift a low equivalence $\sim_L$ to $(\sim_L)^*$ which we simply denote by $\sim_L^*$.

**Definition 4.** Given a security typing $\Gamma$, a probabilistic time bisimilarity $\sim$ is the largest symmetric relation on configurations such that whenever $c_1 \sim c_2$, then $c_1 \Longrightarrow \chi_1$ implies that there exists $\chi_2$ such that $c_2 \Longrightarrow \chi_2$ and $\chi_1 \sim_L^* \chi_2$.

We say that two configurations are probabilistic time bisimilar or PT-bisimilar, $c_1 \sim c_2$, if there exists a probabilistic time bisimilarity relation in which they are related.

We now exploit the notion of bisimilarity introduced above in order to introduce a security property ensuring that a system is confined against any combined attacks based on both timing and probabilistic covert channels.

**Definition 5.** A pWhile program $P$ is *probabilistic time secure* or PT-secure if for any set of initial states $E$ and $E'$ such that $E_L = E'_L$, we have $\langle E, P\rangle \sim \langle E', P\rangle$.

### 5.3 Security Typing

Agat's transformation is based on a (security) typing of commands which is based on the basic (security) typing of variables via $\Gamma$. We only have to add a rule for the **choose** statement (essentially a straight forward extension of the rule for **if**). In detail, we have the following typing rules:

$$(\text{Assign}_H)\ \frac{\Gamma \vdash_{\leq} a : \overline{\tau}_s \quad \Gamma \vdash_{=} x : \overline{\tau}_H \quad s \leq H}{\Gamma \vdash x := a : \mathbf{skipAsn}\ x\ a}$$

$$(\text{Assign}_L)\ \frac{\Gamma \vdash_{\leq} a : \overline{\tau}_L \quad \Gamma \vdash_{=} x : \overline{\tau}_L}{\Gamma \vdash x := a : x := a}$$

$$(\text{Seq})\ \frac{\Gamma \vdash C : C_L \quad \Gamma \vdash D : D_L}{\Gamma C; D : C_L; D_L}$$

$$(\text{If}_H)\ \frac{\Gamma \vdash_{\leq} b : \mathbf{Bool}_H \quad \Gamma \vdash C : C_L \quad \Gamma \vdash D : D_L}{\Gamma \vdash \mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ D : \mathbf{skipIf}\ b\ C_L} \quad C_L \sim D_L$$

$$(\text{If}_L)\ \frac{\Gamma \vdash_{\leq} b : \mathbf{Bool}_L \quad \Gamma \vdash C : C_L \quad \Gamma \vdash D : D_L}{\Gamma \vdash \mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ D : \mathbf{if}\ b\ \mathbf{then}\ C_L\ \mathbf{else}\ D_L}$$

$$(\text{While})\ \frac{\Gamma \vdash_{\leq} b : \mathbf{Bool}_L \quad \Gamma \vdash C : C_L}{\Gamma \vdash \mathbf{while}\ b\ \mathbf{do}\ C : \mathbf{while}\ b\ \mathbf{do}\ C_L}$$

$$(\text{Choose})\ \frac{\Gamma \vdash C : C_L \quad \Gamma \vdash D : D_L}{\Gamma \vdash [\mathbf{choose}]^p\ C\ \mathbf{or}\ D : [\mathbf{choose}]^p\ C_L\ \mathbf{or}\ D_L}$$

$$(\text{SkipAsn})\ \frac{}{\Gamma \vdash \mathbf{skipAsn}\ x\ a : \mathbf{skipAsn}\ x\ a}$$

$$(\text{SkipIf})\ \frac{\Gamma \vdash C : C_L}{\Gamma \vdash \mathbf{skipIf}\ b\ C : \mathbf{skipIf}\ b\ C_L}$$

Note that the rule $(\text{If}_H)$ refers to the *semantic* notion of timed bisimilarity (as introduced before).

### 5.4 A Probabilistic Version of Agat's Transformation

In order to transform programs into secure versions we need to introduce an auxiliary notion, that of the global effect $ge(C)$ of commands. This is used to identify (global) variables which might be changed when a command $C$ is executed – this is closely related to the $\widehat{\mathsf{X}}$ function from the previous section. This is its formal definition:

$$\begin{aligned}
ge(x := a) &= \{x\} \\
ge(C_1; C_2) &= ge(C_1) \ \cup \ ge(C_2) \\
ge(\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2) &= ge(C_1) \ \cup \ ge(C_2) \\
ge(\textbf{while } b \textbf{ do } C) &= ge(C) \\
ge([\textbf{choose}]^p \ C_1 \textbf{ or } C_2) &= ge(C_1) \ \cup \ ge(C_2) \\
ge(\textbf{skipAsn } x \ a) &= \emptyset \\
ge(\textbf{skipIf } b \ C) &= ge(C)
\end{aligned}$$

Finally, based on the security typing the notion of global effects we can specify the rules for "transforming out timing leaks". The judgments or transformation rules are of the general form:

$$\Gamma \vdash C \hookrightarrow D \mid D_L$$

which represents the fact that with a certain (security) typing $\Gamma$ we can transform the statement $C$ into $D$ – we also record as a side-product the so-called *low slice* $D_L$ of $D$. Again all these rules can be found in [12] with the exception of rule ($\text{If}_H$): Here we replace – as indicated before – the branches of an **if** statement not just by the correctly "padded version" as in [12], instead we introduce in every branch a choice such that the secure replacement will be executed only with probability $p$ while with probability $1 - p$ the original (and shorter) code

fragment will be executed.

$$\frac{\Gamma \vdash_{\leq} a : \overline{\tau}_s \quad \Gamma \vdash_{=} x : \overline{\tau}_H \quad s \leq H}{\Gamma \vdash x := a \hookrightarrow x := a \mid \mathbf{skipAsn}\ x\ a}$$

$$\frac{\Gamma \vdash_{\leq} a : \overline{\tau}_L \quad \Gamma \vdash_{=} x : \overline{\tau}_L}{\Gamma \vdash x := a \hookrightarrow x := a \mid x := a}$$

$$\frac{\Gamma \vdash C_1 \hookrightarrow D_1 \mid D_{1L} \quad \Gamma \vdash C_2 \hookrightarrow D_2 \mid D_{2L}}{\Gamma \vdash C_1; C_2 \hookrightarrow D_1; D_2 \mid D_{1L}; D_{2L}}$$

$$\frac{\Gamma \vdash_{\leq} b : \mathbf{Bool}_H \quad \Gamma \vdash C_1 \hookrightarrow D_1 \mid D_{1L} \quad \Gamma \vdash C_2 \hookrightarrow D_2 \mid D_{2L} \quad ge(D_{1L}) = \emptyset \quad ge(D_{2L}) = \emptyset}{\Gamma \vdash \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \hookrightarrow \begin{array}{l}\mathbf{if}\ b\ \mathbf{then}\ ([\mathbf{choose}]^p\ D_1\ \mathbf{or}\ D_1; D_{2L})\ \mathbf{else} \\ ([\mathbf{choose}]^p\ D_2\ \mathbf{or}\ D_{1L}; D_2) \mid \mathbf{skipIf}\ b\ (D_{1L}; D_{2L})\end{array}}$$

$$\frac{\Gamma \vdash_{\leq} b : \mathbf{Bool}_L \quad \Gamma \vdash C_1 \hookrightarrow D_1 \mid D_{1L} \quad \Gamma \vdash C_2 \hookrightarrow D_2 \mid D_{2L}}{\Gamma \vdash \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \hookrightarrow \mathbf{if}\ b\ \mathbf{then}\ D_1\ \mathbf{else}\ D_2 \mid \mathbf{if}\ b\ \mathbf{then}\ D_{1L}\ \mathbf{else}\ D_{2L}}$$

$$\frac{\Gamma \vdash_{\leq} b : \mathbf{Bool}_L \quad \Gamma \vdash C \hookrightarrow D \mid D_L}{\Gamma \vdash \mathbf{while}\ b\ \mathbf{do}\ C \hookrightarrow \mathbf{while}\ b\ \mathbf{do}\ D \mid \mathbf{while}\ b\ \mathbf{do}\ D_L}$$

$$\frac{\Gamma \vdash C_1 \hookrightarrow D_1 \mid D_{1L} \quad \Gamma \vdash C_2 \hookrightarrow D_2 \mid D_{2L}}{\Gamma \vdash [\mathbf{choose}]^p\ C_1\ \mathbf{or}\ C_2 \hookrightarrow [\mathbf{choose}]^p\ D_1\ \mathbf{or}\ D_2 \mid [\mathbf{choose}]^p\ D_{1L}\ \mathbf{or}\ D_{2L}}$$

$$\frac{}{\Gamma \vdash \mathbf{skipAsn}\ x\ a \hookrightarrow \mathbf{skipAsn}\ x\ a \mid \mathbf{skipAsn}\ x\ a}$$

$$\frac{\Gamma \vdash C \hookrightarrow D \mid D_L}{\Gamma \vdash \mathbf{skipIf}\ b\ C \hookrightarrow \mathbf{skipIf}\ b\ D \mid \mathbf{skipIf}\ b\ D_L}$$

## 6 Conclusion

We have briefly introduced semantics-based program analysis and exemplified it in the context of language-based security. In its thirty year existence, program analysis has moved from an academic curiosity to a routine industrial and commercial tool. Examples include the advanced development tools for embedded systems produced by AbsInt and used by major car and electronic goods manufacturers such as BMW, Daimler and Bosch. Another example is the work of Polyspace (recently taken over by The MathWorks) and their work with Airbus on the verification of fly-by-wire software.

Computer Science has seen a growing move towards the use of probabilistic and quantitative models and techniques. This was first apparent in the performance modelling community. Embedded systems is providing further impetus in this direction; Henzinger and Sifakis make an eloquent case for the reappraisal

of the foundations of computing in [15]. They call for a new scientific foundation which "... will systematically and even-handedly integrate computation and physicality ...". They identify the a list of issues that must be dealt with which include:

- computation and physical constraints
- nondeterminism and probabilities
- functional and performance requirements
- qualitative and quantitative analysis, and
- Boolean and real values

We have pioneered a semantics-based approach to program analysis which begins to address some of these issues and we hope that this paper has given a flavour of some of the possibilities. The papers [16, 17] introduce an approximate version of bisimulation and confinement where the approximation can be used as a measure $\varepsilon$ for the information leakage of the system under analysis. The quantity $\varepsilon$ is formally defined in terms of the norm of a linear operator representing the partition induced by the 'minimal' bisimulation on the set of the states of a given system, i.e. the one minimising the observational difference between the system's components. In [18] we show how to compute a non-trivial upper bound $\delta$ to $\varepsilon$ by essentially exploiting the algorithmic solution proposed by Paige and Tarjan [19] for computing bisimulation equivalence. This was already adapted to PTS's in [20], where it was used for constructing a padding algorithm as part of a transformational approach to the timing leaks problem.

Clark, Hunt and Malacaria [21, 22] have done some interesting work using information theoretic ideas as a basis for studying information flow. This general theme of incorporating quantitative and probabilistic techniques into program analysis and reasoning is certain to expand. It remains to be seen which approach will eventually prevail but, whichever does, there is no doubt that the arsenal of techniques will be enrichened and program analysis probably will count!

## References


1. Cousot, P., Cousot, R.: Abstract Interpretation and Applications to Logic Programs. Journal of Logic Programming **13**(2-3) (July 1992) 103–180
2. Mycroft, A.: The theory and practice of transforming call-by-need into call-by-value. In: Symposium on Programming. Volume 83 of Lecture Notes in Computer Science., Springer (1980)
3. Abramsky, S., Hankin, C., eds.: Abstract Interpretation of Declarative Languages. Ellis-Horwood, Chichester, England (1987)
4. Nielson, F., Nielson, H.R., Hankin, C.L.: Principles of Program Analysis. Springer (1999)
5. Di Pierro, A., Wiklicky, H.: Concurrent Constraint Programming: Towards Probabilistic Abstract Interpretation. In: Proceedings of PPDP'00, Montréal, Canada, ACM (2000) 127–138
6. Di Pierro, A., Wiklicky, H.: Measuring the precision of abstract interpretations. In: Proceedings of LOPSTR'00. Volume 2042 of Lecture Notes in Computer Science., Springer Verlag (2001) 147–164

7. Monniaux, D.: Abstract interpretation of probabilistic semantics. In: Proceedings of SAS'00. Volume 1824 of Lecture Notes in Computer Science., Springer Verlag (2000)
8. Clark, D., Hankin, C., Hunt, S.: Information flow for algol-like languages. Comput. Lang. **28**(1) (2002) 3–28
9. Denning, D.: A lattice model of secure information flow. In: Communications of the ACM, ACM (1976) 236–243
10. Smith, G., Volpano, D.: Secure information flow in a multi-threaded imperative language. In: Symposium on Principles of Programming Languages (POPL'98), San Diego, California, ACM (1998) 355–364
11. Kocher, P.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Proceedings of CRYPTO '96. Volume 1109 of Lecture Notes in Computer Science., Springer Verlag (1996) 104–113
12. Agat, J.: Transforming out timing leaks. In: Proceedings of POPL'00, ACM Press (2000) 40–53
13. Larsen, K., Skou, A.: Bisimulation through probabilistic testing. Information and Computation **94** (1991) 1–28
14. Jonsson, B., Yi, W., Larsen, K.: 11. In: Probabilistic Extentions of Process Algebras. Elsevier Science, Amsterdam (2001) 685–710 see [23].
15. Henzinger, T., Sifakis, J.: The discipline of embedded systems design. IEEE Computer (2007)
16. Di Pierro, A., Hankin, C., Wiklicky, H.: Measuring the confinement of probabilistic systems. Theoretical Computer Science **340**(1) (2005) 3–56
17. Di Pierro, A., Hankin, C., Wiklicky, H.: Quantitative relations and approximate process equivalences. In Lugiez, D., ed.: Proceedings of CONCUR'03. Volume 2761 of Lecture Notes in Computer Science., Springer Verlag (2003) 508–522
18. Di Pierro, A., Hankin, C., Wiklicky, H.: Quantifying timing leaks and cost optimisation. In: Proceedings 10th International Conference on Information and Computer Security. Lecture Notes in Computer Science, Springer Verlag (2008) to appear
19. Paige, R., Tarjan, R.: Three partition refinement algorithms. SIAM Journal of Computation **16**(6) (1987) 973–989
20. Di Pierro, A., Hankin, C., Siveroni, I., Wiklicky, H.: Tempus fugit: How to plug it. Journal of Logic and Algebraic Programming **72**(2) (2007) 173–190
21. Clark, D., Hunt, S., Malacaria, P.: Quantitative information flow, relations and polymorphic types. Journal of Logic and Computation **15**(2) (2005)
22. Malacaria, P.: Assessing security threats of looping constructs. In: Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), ACM Press (2007)
23. Bergstra, J., Ponse, A., Smolka, S., eds.: Handbook of Process Algebra. Elsevier Science, Amsterdam (2001)