IJIS manuscript No.
(will be inserted by the editor)

Probabilistic Timing Covert Channels: To Close or not to Close?

Alessandra Di Pierro · Chris Hankin · Herbert Wiklicky

Received: date / Accepted: date

Abstract We develop a new notion of security against timing attacks where the attacker is able to simultaneously observe the execution time of a program and the probability of the values of low variables. We then propose an algorithm which computes an estimate of the security of a program with respect to this notion in terms of timing leakage, and show how to use this estimate for cost optimisation.

Keywords Covert Timing Channels · Probabilistic Programs · Program Transformation · Quantitative Security · Approximate Bisimilarity

1 Introduction

Early work on language-based security, such as Volpano and Smith's type systems [1], 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, 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. [2]). On the other hand, observing the probability distribution of the low events may also offer the possibility of knowing secret information; in this case the leakage forms a *probabilistic covert channel*, which also represents a serious problem for the security of many daily life applications (cf. e.g. [4]).

We develop a new notion of security against timing attacks where the attacker is able to simultaneously observe the execution time of a (probabilistic) program and the probability of the values of low variables. This notion is a non-trivial extension of similar ideas for deterministic programs [3] which also cover attacks based on the combined observation of time and low variables. This earlier work presents an approach which, having identified a covert timing channel, provides a program transformation which neutralises the channel.

We start by introducing a semantic model of timed probabilistic transition systems. Our approach is based on modelling programs essentially as Markov Chains (MC) where the stochastic behaviour is determined by a joint distribution on both the values assigned to the program's variables and the time it takes the program to perform a given command. This is very different from approaches in the area of automata theory which are also dealing with both time and probability. In this area the timed automata constitute a well-established model [6]. These automata have been extended with probability and used in model checking for the verification of probabilistic timed temporal logic properties of real-time systems [7]. The resulting model is however essentially different from our MC model, as it is based on a Markov Decision Process where rewards are interpreted as time durations. In particular, the presence of non-determinism makes it not very appropriate as

A. Di Pierro
University of Verona
Ca' Vignal 2 - Strada le Grazie 15
I-37134 Verona, Italy
E-mail: dipierro@sci.univr.it

C. Hankin
Imperial College London, 180 Queen's Gate
London SW7 2AZ, UK
E-mail: clh@doc.ic.ac.uk

H. Wiklicky
Imperial College London, 180 Queen's Gate
London SW7 2AZ, UK
E-mail: herbert@doc.ic.ac.uk

a base of the quantitative analysis we propose in this paper which aims at measuring timing leaks.

We next present a concrete programming language with a timed probabilistic transition system as its execution model. This language is based on the language studied in [3] but is extended with a probabilistic choice construct – whilst this may not play a role in user programs, it has an essential role in our program transformation. In order to determine and quantify the security of systems and the effectiveness of potential countermeasures against timing attacks, we then discuss an approximate notion of timed bisimilarity and construct an algorithm for computing a quantitative estimate of the vulnerability of a system against timing attacks; this is given in terms of the mismatch between the actual transition probabilities and those of an ideal perfectly confined program. We show the correctness of our algorithm with respect to the notion of security against timing leaks; in particular we show that it returns a zero estimate whenever the program is secure, i.e. no probabilistic timing covert channels can reveal the program's private information.

The security measure we use in this paper stems from process algebraic models [11,29,12,16] where security is identified with indistinguishability in the sense of bisimulation equivalence: A system is safe if for different values of a secret the behaviours are bisimilar. In particular, it refers to the notion of ϵ-bisimilarity originally introduced in [11,29]. Unfortunately, this measure, like other approximate bisimilarity notions – see e.g. [28] – although theoretically appealing appears to be difficult to compute. The concrete measure, which we call δ, computed by our algorithm is therefore only an (easily computable) approximation of this measure obtained by exploiting the fact that the executions of the probabilistic *non-concurrent* language we consider form trees, i.e. acyclic rather than general graphs. However, due to its origins as a 'bisimilarity' notion, the measure δ treats time labels as completely different even if they differ only a little bit. In order to combine 'bisimilarity' and 'similarity' of labels we therefore introduce another notion δ', which re-scale the behavioural differences of two programs registered by the measure δ via a weight expressing some kind of similarity on the labels.

Finally, we present a probabilistic variation of the padding algorithm in [3] which we use to illustrate – via an example – a technique for formally analysing the trade-off between security costs and protection, the latter being expressed via any security measure such as δ or δ'. This variation consists in introducing randomness into the transformation process, so that the program will be effectively transformed only with a certain probability. The percentage of security can be in this way established on the base of the optimisation of other computational costs. This leads to an answer to the question posed in the title which is neither a 'yes' nor a 'not', but rather 'it depends'. In fact, depending on the particular objective one may need to partially close the channel in order to get the best performance.

An earlier version of this paper appeared in the Proceedings of the 10th *International Conference on Information and Communications Security* [5].

2 The Approach: Introducing Noise

In order to illustrate the original approach by Agat as well as our own extension let us consider the following simple program:

```
if k==1 then
  skip(1); skip(10)
else
  skip(1)
fi;
```

We use here a simple "waiting" statement `skip(n)` which has no effect besides that it takes n time units to execute.

Assuming that `k` $\in \{0,1\}$ is a secret, i.e. high, variable we obviously have a problem: If we execute the program just once and it takes 11 time steps, i.e. eleven times the time it takes to perform a semantically meaningless `skip`, then we know *for certain* that the secret value of k was 1; if on the other hand it terminates in just one time step, then we know *for certain* that we executed this fragment in a situation with $k = 0$. There are only two possible running times (1 or 11 times steps) and thus we always know the value of k if we only observe the running time.

Agat's idea now was to simply transform the program in such a way that both branches of the **if** statement have the same running time while their effective semantics stays the same. Unfortunately, it will in general not be possible, except perhaps if we use some program optimisation, to shorten the executions of the longer, `then` branch, thus the general solution is to "pad" the `else` branch by adding 10 additional time steps making the running times of executions for `k==0` and `k==1` indistinguishable while keeping the original semantics of the branches unchanged. Concretely – without discussing the details of his transformation here – Agat replaces the above program by something like:

```
if k==1 then
  skip(1); skip(10)
else
  skip(1); skip(10)
fi;
```

Now the problem is that this introduces a quite substantial overhead: whenever k is 0 we will increase the running time by about an order of magnitude. The issue is whether it is possible to get away with a smaller overhead. We could, for example, try to fix this by a "partial" padding, i.e. adding only some of the missing time to the shorter branch. As an example consider the program

```
if k==1 then
  skip(1); skip(10)
else
  skip(1); skip(5)
fi;
```

Clearly, this adds only half the needed time overhead for $k = 0$. However, this program now is as leaky as the original one: if we observe a running time of 11 time steps we know *for certain* that `k==0`, etc. The only way we could perhaps get away with this approach would be if the difference between the `then` and `else` branch is smaller than the precision of the clock used by an attacker, e.g. 10.9 vs 11 time steps – but in this case the additional overhead is again nearly as bad as in the case of Agat's fix.

Our approach is based on the idea of introducing *noise* in the padding technique in order to obfuscate the running time. One easy way to do this is to make it a random decision (of the program) whether to execute the original or the padded version, let's say with a 50 : 50 chance.

```
if k==1 then
  skip(1); skip(10)
else
  choose 0.5: skip(1)
  or     0.5: skip(1); skip(10)
  ro
fi;
```

Now the average running time of the `else` branch is again increased only by half of what a prefect padding would require. However, if we observe a running time of 11 time steps, it could be that $k = 0$ as well as $k = 1$. Thus, the running time does not allow to determine *with certainty* what the value of k was. Furthermore, each time we even execute the program with the same values for k we get different running times. This is particularly important, as it is usually the case that our little program is executed repeatedly. Consider, for example, our transformed fragment as part of a `for` or `while` loop:

```
i := 1;
while i<=3 do
   if k[i]==1 then
     skip(1); skip(10)
   else
     choose 0.5: skip(1)
     or     0.5: skip(1); skip(10)
     ro
   fi;
od;
```

Let us take the array k being $[1, 0, 0]$. With "partial" padding we would then observe always a running time of $11 + 6 + 6 = 23$ time steps. If, in particular, we could observe each repetition separately we could exactly determine each bit of k. With "probabilistic" padding the running time could be $11 + 1 + 1 = 13$, or $11 + 11 + 1 = 23$, or even $11 + 11 + 11 = 33$ time steps, though in the average the overhead would be the same as with "partial" padding. Furthermore, we can in general not reconstruct the array k, even if we would know the times for every iteration, even less so when we know only the total running time. If the three iterations take 11, 1, and 1 steps respectively, it could be that $k = [1, 0, 0]$ but also $k = [0, 0, 0]$. In the case that all three iterations take 11 time steps each, then we have all possibilities for k.

There are, of course, many more similar ways of introducing "noise" into a program. In the following we will not fix a constant 50 : 50 chance between executing the original program and the fully padded version; instead we will use a parameter p to indicate how much padding, and thus overhead, we are willing to accept. For $p = 1$ we will get, as an extreme case, "perfect" padding. We will however not investigate more complicated schemes, where p could depend on the context, the environment or state, etc.

Obviously, with our "noisy" approach, something can be said about k, i.e. some information about the secret is leaked. For example, with the running times $11, 1, 1$ we can exclude that $k = [1, 0, 1]$, etc. However, our "probabilistic" padding leads to a smaller time overhead than Agat's transformation. This means that the secret is not perfectly protected but only up to a point. Our next aim is to measure the security level our "probabilistic" padding allows for. We will base the "security measure" on the notion of ε-bisimilarity which we introduced in previous work and which has a direct statistical interpretation via so-called hypothesis testing [12].

The important point we aim to demonstrate ultimately in this paper is that by using *a* quantitative concept of security and *a* quantitative notion of security costs one can investigate in a formal way the tradeoff between these quantities. One outcome is the realisation that "perfect" padding a'la Agat is not necessarily an "optimal" padding, but that instead a detailed analysis of the cost/security balance can result in an optimal "probabilistic" padding with $p \neq 1$.

3 The Model

We introduce a general model for the semantics of programs where time and probability are explicitly introduced in order to keep track of both the probabilistic evolution of the program/system state and its running time.

The scenario we have in mind is that of a multilevel security system and an attacker who can observe the system looking at the values of its public variables and the time it takes to perform a given operation or before terminating, or other similar properties related to its timing behaviour.

In order to keep the model simple, we assume that the execution time of a statement is constant and that there is no distinction between any 'local' and 'global' clocks. In a more realistic model, one has – of course – to take into account that the execution speed might differ depending on which other process is running on the same system and/or delays due to uncontrollable events in the communication infrastructure, i.e. network.

Our reference model is the timed probabilistic transition system we define below. The intuitive idea is that of a probabilistic transition system, similar to those defined in all generality in [8], but with transition probabilities defined by a joint distribution of two random variables representing the variable updates and the time, respectively.

Let us consider a finite set X, and let $\mathbf{Dist}(X)$ denote the set of all *probability distributions* on X, that is the set of all functions $\pi : X \to [0,1]$, such that $\sum_{x \in X} \pi(x) = 1$. We often represent these functions as sets of tuples $\{\langle x, \pi(x)\rangle\}_{x \in X}$.

If the set X is presented as a Cartesian product, i.e. $X = X_1 \times X_2$, then we refer to a distribution on X also as a *joint distribution* on X_1 and X_2. A joint distribution associates to each pair (x_1, x_2), with $x_1 \in X_1, x_2 \in X_2$ the probability $\pi(x_1, x_2)$.

It is important to point out that, in general, it is not possible to define any joint distribution on $X_1 \times X_2$ as a 'product' of distributions on X_1 and X_2, i.e. for a given joint distribution π on $X = X_1 \times X_2$ it is, in general, not possible to find distributions π_1 and π_2 on X_1 and X_2 such that for all $(x_1, x_2) \in X_1 \times X_2$ we have $\pi(x_1, x_2) = \pi_1(x_1)\pi_2(x_2)$. In the special cases where a joint distribution π can be expressed in this way, as a 'product', we say that the distributions π_1 and π_2 are *independent* (cf. e.g. [9]).

3.1 Timed Probabilistic Transition Systems

The execution model of programs which we will use in the following is that of a labelled transition system; more precisely, we will consider probabilistic transition systems (PTS). We will put labels on transitions as well as states; the former will have "times" associated with them while the latter will be labelled by uninterpreted entities which are intended to represent the values of (low security) variables, i.e. the computational state during the execution of a program. We will not specify what kind of "time labels" we use, e.g. whether we have a discrete or continuous time model. We just assume that time labels are taken from a finite set $\mathbb{T} \subseteq \mathbb{R}^+$ of positive real numbers. The "state labels" will be taken from an abstract set which we denote by $\mathbb{L}$.

Definition 1 We define a *timed Probabilistic Transition System with labelled states*, or tPTS, as a triple $(S, \longrightarrow, \lambda)$, with S a finite set of *states*, $\longrightarrow \subseteq S \times [0,1] \times \mathbb{T} \times S$ a probabilistic transition relation, and $\lambda : S \to \mathbb{L}$ a state labelling function.

For $(s_1, p, t, s_2) \in \longrightarrow$ we write $s_1 \xrightarrow{p:t} s_2$ with $s_1, s_2 \in S$, $p \in [0,1]$ and $t \in \mathbb{T}$.

A general PTS allows for various forms of nondeterminism; this can be restricted by imposing one of the following conditions (or both):

1. for all $s \in S$ and $t \in \mathbb{T}$ we have
$$\sum_i \{p_i \mid (s, p_i, t, s_i) \in \longrightarrow\} = 1$$
2. for all possible times $t \in \mathbb{T}$ and pairs of states $s_1, s_2 \in S$ there is *at most one* tuple $(s_1, p, t, s_2) \in \longrightarrow$.

The first condition means that we consider a *purely probabilistic* or *generative* execution model. The second condition allows us to associate a unique probability to every transition time between two states, i.e. triple (s_1, t, s_2); this means that we can define a function $\pi : S \times \mathbb{T} \times S \to [0,1]$ such that $s_1 \xrightarrow{p:t} s_2$ iff $\pi(s_1, t, s_2) = p$. Note however, that it is still possible to have differently timed transitions between states, i.e. it is possible to have $(s_1, t_1, p_2, s_2) \in \longrightarrow$ as well as also $(s_1, t_2, p_2, s_2) \in \longrightarrow$ with $t_1 \neq t_2$.

In the case where for all $s_1, s_2 \in S$ there exists at most one $(s_1, t, p, s_2) \in \longrightarrow$, we can also represent a timed Probabilistic Transition System with labelled states as a quadruple $(S, \longrightarrow, \tau, \lambda)$ with $\tau : S \times S \to [0,1] \times \mathbb{T}$, a timing function. Thus, to any two states s_1 and s_2 we associate a unique transition time t_{s_1,s_2} and probability p_{s_1,s_2}.

Both the concrete and the abstract semantics we will define in Section 1 for our imperative language are generative tPTS's; however, while the concrete semantics satisfies condition 2., the abstract semantics does not.

Definition 2 Consider a tPTS $(S, \longrightarrow, \lambda)$ and an *initial state* $s_0 \in S$. An *execution sequence* or *trace* starting in s_0 is a sequence $(s_0, s_1, \ldots)$ such that $s_i \xrightarrow{p_i:t_i} s_{i+1}$, for all $i = 0, 1, 2, \ldots$.

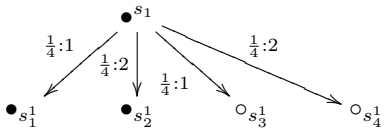
We associate, in the obvious way, to an execution sequence $\sigma = (s_0, s_1, \ldots)$ three more sequences: (i) the transition probability sequence: $(p_1, p_2, \ldots)$, (ii) a time stamp sequence: $(t_1, t_2, \ldots)$, and (iii) a state label sequence: $(\lambda(s_o), \lambda(s_1), \ldots)$.

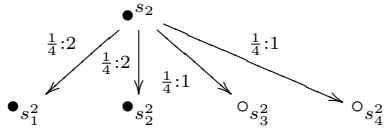
Even for a tPTS with a finite number of states it is possible to have infinite execution sequences. It is thus, in general, necessary to consider measure theoretic notions in order to define a mathematically sound model for the possible behaviours of a tPTS. However, as long as we consider only terminating systems, i.e. finite traces, things are somewhat simpler. In particular, in this case, probability distributions can replace measures as they are equivalent. In this paper we restrict our attention to terminating traces and probability distributions. This allows us to define for every finite execution sequence $\sigma = (s_0, s_1, \ldots)$ its *running time* as $\tau(\sigma) = \sum t_i$, and its *execution probability* as $\pi(\sigma) = \prod t_i$. We will also associate to every state s_0 its *execution tree*, i.e. the collection of all execution sequences starting in s_0.

3.2 Observing tPTS's

In Section 4 we will present an operational semantics of a simple imperative programming language, pWhile, via a tPTS. Based on this model we will then investigate the vulnerability against attackers who are able to observe (i) the time, and (ii) the state labels, i.e. the low variables. In this setting we will argue that the combined observation of time and low variables is more powerful than the observation of time and low variables separately. The following example aims to illustrate this aspect which comes from the properties of joint probability distributions.

Example 1 In order to illustrate the role of joint distributions in the observation of timed PTS's let us consider the following two simple systems:

$A =$ 

$B =$ 

We assume that the attacker can observe the execution times and that he/she is also able to (partially) distinguish (the final) states. In our example we assume that the states depicted as $\bullet$ and $\circ$ form two classes which the attacker can identify (e.g. because $\bullet$ and $\circ$ states have the same values for low variables). The question now is whether this information allows the attacker to distinguish the two tPTS's.

If we consider the information obtained by observing the running time, we see that both systems exhibit the same time behaviour corresponding to the distribution $\{\langle 1, \frac{1}{2}\rangle, \{\langle 2, \frac{1}{2}\rangle\}$ over $\mathbb{T} = \{1, 2\}$. The same is true in the case where the information is obtained by inspecting the final states: we have the distributions $\{\langle \bullet, \frac{1}{2}\rangle, \langle \circ, \frac{1}{2}\rangle\}$ over $\mathbb{L} = \{\bullet, \circ\}$ for both systems.

However, considering that the attacker can observe running time and labels *simultaneously*, we see that the system A always runs for 2 time steps iff it ends up in a $\bullet$ state and 1 time step iff it ends up in a $\circ$ state. In system B there is no such *correlation* between running time and final state. The difference between the two systems, which allows an attacker to distinguish them, is reflected in the joint distributions over $\mathbb{T} \times \mathbb{L}$.

These joint probability distributions can be expressed in matrix form for the two systems above as:

$\chi_1(t,l)$	1	2
$\bullet$	$\frac{1}{4}$	$\frac{1}{4}$
$\circ$	$\frac{1}{4}$	$\frac{1}{4}$

$\chi_2(t,l)$	1	2
$\bullet$	0	$\frac{1}{2}$
$\circ$	$\frac{1}{2}$	0

Note that while χ_1 is the product of two *independent* probability distributions on $\mathbb{T}$ and $\mathbb{L}$ it is not possible to represent χ_2 in the same way.

4 An Imperative Language

We consider a language similar to one used in [3] with the addition of a probabilistic *choice* construct. We omit from Agat's language *output* commands and *let* statements, i.e. local evaluation environments, as well as data structures like *array*s and *record*s.

The syntax of the language is as follows:

Operators:	op	$::= + \mid * \mid - \mid = \mid < \mid \ldots$
Expressions:	e	$::= v \mid x \mid e\ op\ e$
Commands:	C, D	$::= x := e \mid \textbf{skipAsn}\ x\ e$
		$\mid \textbf{if}\ (e)\ \textbf{then}\ C\ \textbf{else}\ D$
		$\mid \textbf{skipIf}\ e\ C \mid \textbf{while}\ (e)\ \textbf{do}\ C$
		$\mid C; D \mid \textbf{choose}^p\ C\ \textbf{or}\ D$
Basic Values:	v	$::= n \mid \textbf{true} \mid \textbf{false}$

As explained in [3] and shown later, the skip commands (**skipAsn** and **skipIf**) are used in the transformation to pad programs; as can be seen from the semantics they exhibit the same timing behaviour as assignment and the conditional, respectively. The probabilistic choice is used in an essential way in the program transformation presented later. Note that $p \in [0, 1]$ in $\textbf{choose}^p\ C\ \textbf{or}\ D$ is a constant. We will distinguish purely on syntactic grounds between **deterministic** programs which do not contain a *choice* construct and **probabilistic** ones which do. It is easy to see that deterministic programs in our language form a proper sub-language of the one in [3].

We also keep the language of types as in [3], although in a simplified form:

Security levels	s	$::= \mathrm{L} \mid \mathrm{H}$
Base types	$\overline{\tau}$	$::= \textbf{Int} \mid \textbf{Bool}$
Security types	τ	$::= \overline{\tau}_s$

with $\mathrm{L} \leq \mathrm{H}$ and $s \leq s$ and sub-typing relation:

$$\frac{s_1 \leq s_2}{\overline{\tau}_{s_1} \leq \overline{\tau}_{s_2}}.$$

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 call a pair $\langle E \mid C \rangle$, of a state and a command a *configuration*. 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 for configurations the shorthand notation $c, c_1, c_2, \ldots, c', c'_1, \ldots$. We will also denote by **Conf** the set of all configurations.

4.1 Operational Semantics

The operational semantics of pWhile – except for the probabilistic choice construct – follows essentially the one presented in [3]. For the convenience of the reader we present here all the rules which are based on the big step semantics for expressions (where $[\![op]\!]$ represents the usual semantics of arithmetic and Boolean operators):

$$E \vdash v \Downarrow v \qquad \frac{E(x) = v}{E \vdash x \Downarrow v} \qquad \frac{E \vdash e_1 \Downarrow v_1 \qquad E \vdash e_2 \Downarrow v_2}{E \vdash e_1\ op\ e_2 \Downarrow v_1 [\![op]\!] v_2}$$

The small step semantics is then defined as a timed PTS via the SOS rules in Table 1.

The time labels $t_{\cdot}$ represent the time it takes to perform certain operations: t_x is the time to store a variable, t_e is the time it takes to evaluate an 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. By ts we denote any sequence of time labels and with $\surd$ we indicate termination.

The rule (Choose) is the only new rule with respect to the original semantics in [3]. It 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. This rule together with the standard transition rules for the other constructs of the language define a tPTS $(S, \longrightarrow, \lambda)$ for our pWhile language according to Definition 1. In this tPTS, $\longrightarrow$ is the transition relation $\xrightarrow{p:t}$ defined in Table 1 (where the time t is, strictly speaking, the accumulated time labels, i.e. $t = \sum ts$), the state labels are given by the environment, i.e. $\lambda(\langle E \mid C \rangle) = E$, and the elements in S are configurations, i.e. $S = \textbf{Conf}$. In the rest of the paper we will use interchangeably the words 'state' and 'configuration' to indicate such elements.

As mentioned earlier the rules (SkipAsn) and (SkipIf) are skip operations which have the same timing behaviour as the assignment and conditional, respectively. The rules show why the skip operations need to be parameterised: in order to access the timing behaviour of the critical subcomponents of the associated command.

4.2 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 σ *deterministic* if $\pi(\sigma) = 1$, and we call it *low stable* if $\lambda(c_i) =_L \lambda(c_j)$ for all $c_i, c_j \in \sigma$. 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.

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

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

(If) $$\frac{E \vdash e \Downarrow \mathbf{true}}{\langle E \mid \mathbf{if}\ (e)\ \mathbf{then}\ C\ \mathbf{else}\ D\rangle \xrightarrow{1:t_e \cdot t_{br}} \langle E \mid C\rangle} \qquad \frac{E \vdash e \Downarrow \mathbf{false}}{\langle E \mid \mathbf{if}\ (e)\ \mathbf{then}\ C\ \mathbf{else}\ D\rangle \xrightarrow{1:t_e \cdot t_{br}} \langle E \mid D\rangle}$$

(While) $$\frac{E \vdash e \Downarrow \mathbf{false}}{\langle E \mid \mathbf{while}\ (e)\ \mathbf{do}\ C\rangle \xrightarrow{1:t_e \cdot t_{br} \cdot \surd} E} \qquad \frac{E \vdash e \Downarrow \mathbf{true}}{\langle E \mid \mathbf{while}\ (e)\ \mathbf{do}\ C\rangle \xrightarrow{1:t_e \cdot t_{br}} \langle E \mid C; \mathbf{while}\ (e)\ \mathbf{do}\ C\rangle}$$

(Choose) $$\frac{}{\langle E \mid \mathbf{choose}^p\ C\ \mathbf{or}\ D\rangle \xrightarrow{p:t_{ch}} \langle E \mid C\rangle} \qquad \frac{}{\langle E \mid \mathbf{choose}^p\ C\ \mathbf{or}\ D\rangle \xrightarrow{(1-p):t_{ch}} \langle E \mid D\rangle}$$

(SkipAsn) $$\frac{E \vdash e \Downarrow v}{\langle E \mid \mathbf{skipAsn}\ x\ e\rangle \xrightarrow{1:t_e \cdot t_x \cdot t_{asn} \cdot \surd} E}$$

(SkipIf) $$\frac{E \vdash e \Downarrow v \in \{\mathbf{true}, \mathbf{false}\}}{\langle E \mid \mathbf{skipIf}\ e\ C\rangle \xrightarrow{1:t_e \cdot t_{br}} \langle E \mid C\rangle}$$

Table 1 Operational Semantics

Definition 3 We define the collapsed transition relation by:

$\langle E_1 \mid C_1\rangle \overset{p:T}{\Longrightarrow} \langle E_2 \mid C_2\rangle$

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) the (possibly empty) path

$\langle E_1' \mid C_1'\rangle \xrightarrow{1:t_1} \ldots \langle E_2' \mid C_2'\rangle \xrightarrow{1:t_n} \langle E_2 \mid C_2\rangle$

is *deterministic*, if the path is empty then $\langle E_1' \mid C_1'\rangle = \langle E_2' \mid C_2'\rangle = \ldots = \langle E_2 \mid C_2\rangle$,

(iii) the (possibly partially empty) path

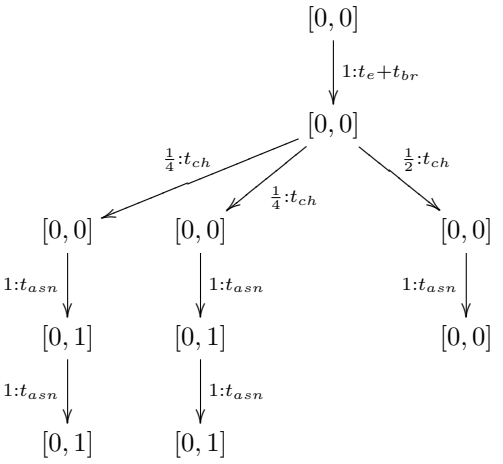
$\langle E_1 \mid C_1\rangle \xrightarrow{p:t} \langle E_1' \mid C_1'\rangle \xrightarrow{1:t_1} \ldots \langle E_2' \mid C_2'\rangle$

is *maximal low stable*, $\langle E_1 \mid C_1\rangle$ and $\langle E_1' \mid C_1'\rangle$ are distinct configurations – cf. (i) – the remaining configurations could be identical to $\langle E_1' \mid C_1'\rangle$ – cf. (ii),

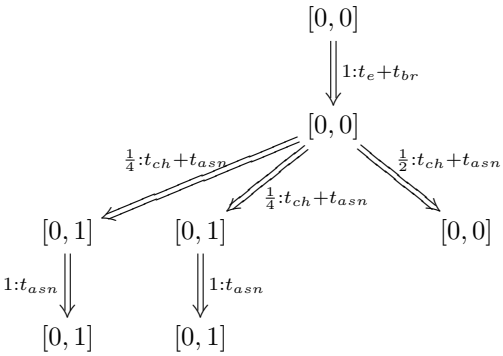
(iv) and $T = t + \sum_{i=1}^{n} t_i$.

Notice that in this definition deterministic (sub-)paths (in case (ii) and (iii)) could be empty, therefore it is not required that $\langle E_1' \mid C_1'\rangle$, $\langle E_2' \mid C_2'\rangle$, ... and $\langle E_2 \mid C_2\rangle$ are distinct.

Definition 3 is illustrated in the following example. In the depicted execution trees we indicate in the nodes only the state and omit the program parts of the corresponding configurations. Moreover, we use the notation $[n, m]$ for the state E where h has value n and l has value m.



becomes



The collapsed execution tree represents in effect what an attacker can actually observe during the program ex-

ecution (for our analysis of the situation we still record the value of h although it is invisible to the attacker).

Note that the collapsed transition relation $\stackrel{p:t}{\Longrightarrow}$ on **Conf** can equivalently be seen as a relation $\Longrightarrow \in (\mathbf{Conf}, \mathbf{JDist})$ between configurations to joint distributions $\chi \in \mathbf{JDist} = \mathbf{Dist}(\mathbb{T} \times \mathbf{Conf})$, that is distributions on time $t \in \mathbb{T}$ and states $s \in \mathbf{Conf}$ such that $\chi(c', t) = p$ iff $c \stackrel{p:t}{\Longrightarrow} c'$.

5 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 (cf. Section 3). A naive probabilistic extension of the Γ-bisimulation notion introduced in [3] 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 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, according to which 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.

5.1 Probabilistic Time Bisimulation

Probabilistic bisimulation as it was first introduced in [10] 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 as follows (cf [8, Thm 1]):

$$\mu \sim^* \nu \text{ iff } \forall [s] \in S/_\sim : \mu([s]) = \nu([s]),$$

where for any $\delta \in \mathbf{Dist}(S)$,

$$\delta([s]) = \sum_{s' \in [s]} \delta(s').$$

It follows that $\sim^*$ is also an equivalence relation ([8, 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 $\sim_L^*$.

We define below a probabilistic bisimulation relation where we identify two configurations, c_1 and c_2, if their stepwise behaviour, which we express via the abstract semantics $\Longrightarrow$ introduced in Definition 3, is the 'same'; this means that the joint probability of reaching in a given time low-equivalent configurations from both c_1 and c_2 is the same.

Definition 4 Given a set oL of low variables, a probabilistic time bisimulation $\sim$ is a relation on configurations such that whenever $c_1 \sim c_2$, then $c_1 \Longrightarrow \chi_1$ implies that there exists χ_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 bisimulation relation in which they are related. PT-bisimilarity is an equivalence relation corresponding to the largest PT-bisimulation on configurations.

This definition generalises the one in [3] which only applies to deterministic transition systems. Note that there is a difference between $\sim_L^* = (\sim_L)^*$ and $(\sim^*)_L$. To see this, consider again Example 1. On one hand, if we consider a PT-bisimulation $\sim$ on the states of the two PTS's A and B, we obtain only two classes, the roots $\times = \{s_1, s_2\}$ and the set of leaves $\perp = \{s_j^i\}$ with $i \in \{1, 2\}$ and $j \in \{1, 2, 3, 4\}$. Then $s_1 \Longrightarrow \chi_1$ and $s_2 \Longrightarrow \chi_2$ induce on these two classes the distributions:

$\chi_1(t,c)$	1	2
$\times$	0	0
$\perp$	$\frac{1}{2}$	$\frac{1}{2}$

$\chi_2(t,c)$	1	2
$\times$	0	0
$\perp$	$\frac{1}{2}$	$\frac{1}{2}$

These distributions are therefore identified by the lifting $\sim^*$. We now have to extend this relation over distributions to a low equivalence in order to define $(\sim^*)_L$. We do not define a low equivalence for distributions (we will not need it in the following), however, it is clear that in whatever way we would do this, any distinction between χ_1 and χ_2 will depend at best on the definition of the low equivalence and not on the structure original transition system, and will therefore have to inherit the indistinguishability of χ_1 and χ_2.

On the other hand, the equivalence on joint distributions we need in Definition 4 is the lifting of the low probabilistic time bisimilarity $\sim_L$ expressing the condition that for all classes $[c] \in \mathbf{Conf}/_{\sim_L}$, that is $[c] = \{c' \mid c \sim c' \text{ and } c' =_L c\}$, we have that $\chi_1([c]) = \chi_2([c])$.

In our example this relation leads to a partitions of states as follows:

$\times = \{s_1, s_2\}$
$\bullet = \{s_1^1, s_2^1, s_1^2, s_2^2\}$
$\circ = \{s_3^1, s_4^1, s_3^2, s_4^2\}$

If we lift $s_1 \Longrightarrow \chi_1$ and $s_2 \Longrightarrow \chi_2$ from distributions over states to distribution over classes in $\sim_L$ we get the distributions

$\chi_1(t,l)$	1	2
$\times$	0	0
$\bullet$	$\frac{1}{4}$	$\frac{1}{4}$
$\circ$	$\frac{1}{4}$	$\frac{1}{4}$

$\chi_2(t,l)$	1	2
$\times$	0	0
$\bullet$	0	$\frac{1}{2}$
$\circ$	$\frac{1}{2}$	0

which establishes the fact, we already discussed before, that s_1 and s_2 are not bisimilar according to $(\sim_L)^*$. Only this latter notion is therefore able to take into account the correlation between time and low variables, while the former would be a straightforward generalisation of the time bisimulation in [3] which is unable to model such a correlation.

Definition 5 Two tPTS T_1 and T_2 are PT-bisimilar iff there exists a probabilistic time bisimulation $\sim$ on $\mathbf{Conf}_{T_1} \cup \mathbf{Conf}_{T_2}$ such that for all equivalence classes $[c]_\sim$ we have $[c]_\sim \cap \mathbf{Conf}_{T_1} \neq \emptyset$ and $[c]_\sim \cap \mathbf{Conf}_{T_2} \neq \emptyset$.

5.2 Probabilistic Time Secure Programs

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 6 A pWhile program P is *probabilistic time secure* or PT-secure if for any set of initial states E and E' such that $E_L = E'_L$, the execution trees rooted in $\langle E, P\rangle$ and $\langle E', P\rangle$ are PT-bisimilar.

For deterministic programs, there is always only one execution path if we start in an initial configuration $\langle E, P\rangle$. The only way to obtain truly branching execution trees is via (probabilistic) **choose** statements. The conditional statements, i.e. **if** and **while**, do not introduce branches. In fact, given a concrete configuration, e.g. $\langle E, \textbf{if } (e) \textbf{ then} \ldots\rangle$, we always know how to continue because we know whether the guard expression e evaluates in the environment E into **true** or **false**.

It is easy to see that two execution paths are PT-bisimilar if and only if they have the same execution time (or length) for sections during which the observable low variables do not change. To illustrate this consider the following example.

Example 2 Let us consider just one high variable h and a low variable l which both can take only values in $\{0, 1\}$. The program we wish to investigate (using a concrete syntax which is slightly easier to parse automatically) is then the following:

```
if h==0 then
  h := 1; l := 1
else
  skipAsn h 1; l := 1
fi;
l := 0
```

The execution of this program leads to four possible deterministic paths, depending on the values of h and l. If we only record the values of h and l along these paths by the pair $[h, l]$ we get:

$$[0,0] \xrightarrow{t_{h==0}\cdot t_{br}} [0,0] \xrightarrow{t_1\cdot t_h\cdot t_{asn}} [1,0] \xrightarrow{t_1\cdot t_l\cdot t_{asn}} [1,1] \xrightarrow{t_0\cdot t_l\cdot t_{asn}} [1,0]$$

$$[1,0] \xrightarrow{t_{h==0}\cdot t_{br}} [1,0] \xrightarrow{t_1\cdot t_h\cdot t_{asn}} [1,0] \xrightarrow{t_1\cdot t_l\cdot t_{asn}} [1,1] \xrightarrow{t_0\cdot t_l\cdot t_{asn}} [1,0]$$

$$[0,1] \xrightarrow{t_{h==0}\cdot t_{br}} [0,1] \xrightarrow{t_1\cdot t_h\cdot t_{asn}} [1,1] \xrightarrow{t_1\cdot t_l\cdot t_{asn}} [1,1] \xrightarrow{t_0\cdot t_l\cdot t_{asn}} [1,0]$$

$$[1,1] \xrightarrow{t_{h==0}\cdot t_{br}} [1,1] \xrightarrow{t_1\cdot t_h\cdot t_{asn}} [1,1] \xrightarrow{t_1\cdot t_l\cdot t_{asn}} [1,1] \xrightarrow{t_0\cdot t_l\cdot t_{asn}} [1,0]$$

If we are recording only the observable low variable l these become:

$$[0] \xrightarrow{t_{h==0}\cdot t_{br}} [0] \xrightarrow{t_1\cdot t_h\cdot t_{asn}} [0] \xrightarrow{t_1\cdot t_l\cdot t_{asn}} [1] \xrightarrow{t_0\cdot t_l\cdot t_{asn}} [0]$$

$$[0] \xrightarrow{t_{h==0}\cdot t_{br}} [0] \xrightarrow{t_1\cdot t_h\cdot t_{asn}} [0] \xrightarrow{t_1\cdot t_l\cdot t_{asn}} [1] \xrightarrow{t_0\cdot t_l\cdot t_{asn}} [0]$$

$$[1] \xrightarrow{t_{h==0}\cdot t_{br}} [1] \xrightarrow{t_1\cdot t_h\cdot t_{asn}} [1] \xrightarrow{t_1\cdot t_l\cdot t_{asn}} [1] \xrightarrow{t_0\cdot t_l\cdot t_{asn}} [0]$$

$$[1] \xrightarrow{t_{h==0}\cdot t_{br}} [1] \xrightarrow{t_1\cdot t_h\cdot t_{asn}} [1] \xrightarrow{t_1\cdot t_l\cdot t_{asn}} [1] \xrightarrow{t_0\cdot t_l\cdot t_{asn}} [0]$$

and after collapsing the transition relation we have, if we start with $l \mapsto 0$

$$[0] \xRightarrow{t_{h==0}\cdot t_{br}\cdot t_1\cdot t_h\cdot t_{asn}\cdot t_1\cdot t_l\cdot t_{asn}} [1] \xRightarrow{t_0\cdot t_l\cdot t_{asn}} [0]$$

$$[0] \xRightarrow{t_{h==0}\cdot t_{br}\cdot t_1\cdot t_h\cdot t_{asn}\cdot t_1\cdot t_l\cdot t_{asn}} [1] \xRightarrow{t_0\cdot t_l\cdot t_{asn}} [0]$$

and for $l \mapsto 1$:

$$[1] \xRightarrow{t_{h==0}\cdot t_{br}\cdot t_1\cdot t_h\cdot t_{asn}\cdot t_1\cdot t_l\cdot t_{asn}\cdot t_0\cdot t_l\cdot t_{asn}} [0]$$

$$[1] \xRightarrow{t_{h==0}\cdot t_{br}\cdot t_1\cdot t_h\cdot t_{asn}\cdot t_1\cdot t_l\cdot t_{asn}\cdot t_0\cdot t_l\cdot t_{asn}} [0]$$

Clearly, the pairs of "trees" which start with the same (observable) value for l are PT-bisimilar. Thus we can conclude that our little program is indeed PT-secure.

In the presence of **choose** statements, then we need to consider the branching structure, instead of just the total execution times along collapsed transitions.

Example 3 Take again a high variable h and a low variable l with values in $\{0,1\}$. Then the following (branching) program is *not* PT-secure:

```
if h==0 then
  choose 0.5: h := h; l := 1
  or     0.5: l := 0
  ro
else
  choose 0.5: h := h; l := 0
  or     0.5: l := 1
  ro
fi;
l := 0
```

The detailed construction of the execution trees is rather involved and we will not depict it here. It suffices to say that for $h \mapsto 0$ and $h \mapsto 1$ we essentially get execution trees of the same structure as in Section 3.2.

Let us assume that we start, for example, with the observable state in which $l \mapsto 0$. Then the two execution trees – for $h \mapsto 0$ and $h \mapsto 1$ – are not PT-bisimilar. Thus we come to the conclusion that this program is not PT-secure. This is consistent with what we said about the simultaneous observation of time and final results in Section 3.2.

6 Computing Approximate Bisimulations

Our next task is to quantify the security of a system. Clearly, the prevailing idealist's approach is to distinguish only between the realms of the good and bad, i.e. whether a system is resisting against (all possible) attacks or not. A more economical and practical approach aims instead on distinguishing between the many shades of gray represented by real systems which might be more or less well protected against attacks.

One can think of various ways of measuring the security of programs and we do not aim to provide here a definitive answer, instead we will consider a reasonable measure which allows us to investigate the trade-off between security and security costs. Security measures can be based on a statistical evaluation of case studies, i.e. empirical testing, or introduced via information theoretic notions as in e.g. [34,14,33], or by quantifying the (bi)similarity of systems. In this paper we will follow the latter approach. We will exploit the well-established relationship between indistinguishability and security: A secret is well protected if the observable behaviour of a system which accesses this secret is always the same irrespectively of the concrete value of the secret in question. It is then impossible for an attacker – with certain observational capabilities – to determine the secret by observing the behaviour of the system.

In concurrency theory the notion of *bisimulation* – in all its different timed, barbed, probabilistic, etc. variations – is arguably the most important concept describing process equivalence. It is quite common to define security by reference to this concept by assessing a secret as safe or secure if for two different values of the secret the systems are bisimilar. It is also helpful for our programme that Agat's work [3] too is based on this notion, though given that he only considers deterministic programs it is somewhat degenerated as all his execution trees are non-branching, i.e. are represented by a single execution path.

In this section we will first summarise our notion of *approximate* or ε-bisimulation which we introduced previously [12,13] in order to define a quantified version of security (cf 6.1). It turns out that it is relatively expensive to compute the exact value of ε and we thus introduce a computational approximation δ and discuss the relation between the two quantities δ and ε, in 6.2. Finally, as it turns out that δ (as well as ε) is rather unforgiving regarding to what one might consider only a 'small' behavioural difference, we introduce a weighted version δ' which will use in order to quantify the security of a program (cf 6.4).

6.1 ε-Bisimulation

In [12,13] we introduce an approximate version of bisimulation and confinement where the approximation can be used as a measure ε for the information leakage of the system under analysis. In the above mentioned work we represented systems by linear operators, i.e. by their transition matrices $\mathbf{M}$. In the case of probabilistic programs and systems these matrices $\mathbf{M}$ are the usual well known stochastic matrices which are the generators of the corresponding Markov chains (for details see [12, 13]).

One can then show that two systems $\mathbf{M}_1$ and $\mathbf{M}_2$ are bisimilar if there exist simplified or abstracted versions of $\mathbf{M}_1$ and $\mathbf{M}_2$, represented by matrices $\mathbf{M}_1^\#$ and $\mathbf{M}_2^\#$, such that $\mathbf{M}_1^\# = \mathbf{M}_2^\#$. The abstract systems are obtained by *lumping* states, i.e. by identifying each concrete state s_i with a class C_j of states which are all behavioural equivalent to each other.

Concretely, we can compute this via $n \times m$ matrices $\mathbf{K}$ (where n is the number of concrete states and m the number of abstract classes) with $\mathbf{K}_{ij} = 1$ iff $s_i \in C_j$ and 0 otherwise. We refer to such matrices which have exactly one entry 1 in each row while all other entries are 0 as *classification matrices*, and denote the set of all classification matrices by $\mathcal{K}$. The abstract systems are then given by $\mathbf{M}_i^\# = \mathbf{K}_i^\dagger \mathbf{M}_i \mathbf{K}_i$ with $\mathbf{K}_i$ some classification matrix and $\dagger$ constructing the so called *Moore-*

Penrose pseudo-inverse – in the case of classification matrices $\mathbf{K}^\dagger$ can be constructed as the row-normalised transpose of $\mathbf{K}$.

The problem of showing that two systems $\mathbf{M}_1$ and $\mathbf{M}_2$ are behaviourally equivalent, i.e. are (probabilistically) bisimilar, is now translated into finding two classification matrices $\mathbf{K}_i \in \mathcal{K}$ such that

$$\mathbf{M}_1^{\#} = \mathbf{K}_1^\dagger \mathbf{M}_1 \mathbf{K}_1 = \mathbf{K}_2^\dagger \mathbf{M}_2 \mathbf{K}_2 = \mathbf{M}_2^{\#}.$$

In case that two systems are not bisimilar we still can define a quantity ε which describes how (non-)bisimilar the two systems are. This ε 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 (see again [12] for further details, in particular regarding labelled PTS's):

Definition 7 Let $\mathbf{M}_1$ and $\mathbf{M}_2$ be the matrix representations of two probabilistic transition systems. We say that $\mathbf{M}_1$ and $\mathbf{M}_2$ are *ε-bisimilar*, denoted by $\mathbf{M}_1 \sim_b^\varepsilon \mathbf{M}_2$, iff

$$\inf_{\mathbf{K}_1, \mathbf{K}_2 \in \mathcal{K}} \|\mathbf{K}_1^\dagger \mathbf{M}_1 \mathbf{K}_1 - \mathbf{K}_2^\dagger \mathbf{M}_2 \mathbf{K}_2\| = \varepsilon$$

where $\|.\|$ denotes an appropriate norm, e.g. the supremum norm $\|.\|_\infty$.

In principle, ε provides now also a measure for the security of systems, concretely it says how *confined* (the secret or identity of) a system is. In [12] we investigated this further and provided a statistical interpretation of ε based on *hypothesis testing*: The smaller ε is the more observations by the attacker are needed before she can identify with a certain confidence α the secret of a given process. However, there is a drawback to our construction: We did not provide a computationally feasible way to obtain the inf over all classification matrices. A brute force approach is prohibitively expensive. This is in contrast to the situation in the classical (i.e. non-probabilistic) case where we know of efficient lumping algorithms, like the ones based on the work by Paige and Tarjan [15], which allow for a quasi directed search of the optimal class structure.

6.2 Constructing a Lumping and δ

Already in [16] we exploited the algorithmic solution proposed by Paige and Tarjan [15] for computing bisimulation equivalence over sets and adapted it to probabilistic transition systems. This was in order to introduce a padding algorithm which – contrary to what we aim to achieve in this current paper – attempted to the elimination of timing leaks by transforming the computational paths of a program so as to make it perfectly secure. The resulting transformation was shown to be correct in the sense of preserving the program's I/O behaviour, while eliminating any possible timing covert channel.

The algorithm we present here can be seen as a 'more computational' version of the algorithm in [16] in two ways. First, the abstract labels are replaced by the statements in a concrete language (pWhile) and their execution times; and second, instead of transforming the execution trees, our algorithm accumulates the information about the difference between their transition probabilities and uses this information to compute an upper bound δ to the maximal information leakage ε of the given program.

The algorithmic paradigm for partition refinement introduced by Paige and Tarjan in [15], see also [17,18], constructs a partition of a state space Σ which is *stable* for a given transition relation $\rightarrow$, that is it does not need further refinement. It is a well-known result that this partition corresponds to a bisimulation equivalence on the transition system $(\Sigma, \rightarrow)$. The refinement procedure used in the algorithm consists in *splitting* the blocks in a given partition P by replacing each block $B \in P$ with $B \cap preS$ and $B \setminus preS$, where for any $X \subseteq \Sigma$, $pre(X) = \{s \in \Sigma \mid s \rightarrow x \text{ for some } x \in X\}$. This algorithm has been adapted in [17] to probabilistic processes; in particular, the Derisavi et al. algorithm constructs the optimal lumping quotient of a finite Markov chain, and can then be used to check the probabilistic bisimilarity of two PTS's.

In order to check whether two execution trees T_1 and T_2 in our tPTS model are PT-bisimilar, we can apply a similar refinement technique to the set of states formed by the disjoint union, $T_1 \oplus T_2$, of the states in T_1 and T_2. The lumping procedure starts from an initial partition and refine it step by step by using a *splitter* block B in the following way: Take all the blocks B_i in P and determine for all states in each of these blocks B_i the probabilities and times to reach the splitter block B; if the states in B_i can reach B in a given time with different probabilities, or if the states in B_i do not agree on the values of the low variables then split B_i into subblocks.

Our lumping algorithm QLumping(T_1, T_2) essentially follows the same strategy but, additionally it exploits the 'layering' implicit in a tree structure. Informally, it can be described as follows: It begins with an initial partition of the union $T_1 \oplus T_2$ of the two execution trees, whose blocks are represented by each layer L_i, i.e. set of configurations at the same height (or level) i. We then traverse the trees layer by layer starting from the

leaves, i.e. layer 0. On each layer we refine the partition by a two phases splitting: (i) we group configurations based on the values of the low variables, and (ii) refine the partitioning of the current layer – except in the case of the leave layer – on the basis of the possible transitions of the configurations in the current layer. The idea is to refine a given layer partition until it is stable, i.e. no further sub-partioning is necessary, and then move on to the next layer above.

The sub-partitioning of a layer partition based on the transitions which lead to the layers below can be done in two ways: Either by considering the pre-image of the blocks in the layers below [backward splitting], or by considering the joint distributions of reaching the blocks below from a given configuration in the current layer [forward splitting].

Backward Splitting. We define for every possible time $t \in \mathbb{T}$ and probability $p \in [0,1]$ and any set B of configurations (usually B will be a block) its pre-image

$$pre_p^t(B) = \{c \mid c \xrightarrow{p_i:t} c_i, c_i \in B, \sum p_i = p\}$$

i.e. the set of all configurations c which can reach in time t the block B with probability p. In backward splitting we take a block B_i, called the splitter, in one of the layers below the current layer and check whether one of the blocks B in the current layer is "intersected" by the pre-image of B_i, i.e. if there exists a p and t such that $pre(B_i) \cap B$ is neither empty nor all of B. In this case we split/refine B accordingly.

Forward Splitting. For a given configuration c we denote the joint distribution on $\mathbb{T} \times \mathbf{Conf}(T_1 \oplus T_2)$ of reaching blocks B from c in time t as:

$$\chi_c(B,t) = \sum\{p \mid c \xrightarrow{p:t} c_k, c_k \in B\}.$$

We say that $\chi_c = \chi_{c'}$ if $\chi_c(B,t) = \chi_{c'}(B,t)$ for all block B and times t. In forward splitting we separate configurations c and c' (i.e. put them in new sub-blocks) whenever $\chi_c \neq \chi_{c'}$ – where we actually have to consider only the blocks B_i below the current layer and a finite number of possible transition times.

While backward splitting is conceptually perhaps easier and more in line with the original Paige-Tarjan algorithm, we have chosen here the second approach as it turned out to be easier to implement.

The Procedure QLumping. Algorithm 1 describes essentially the procedure that we have implemented in Octave for computing the approximation δ. It constructs a lumping by using the forward splitting strategy described above, and also computes δ by using the sub-routine CompDelta(L_1, L_2) which returns the best match for each layer. This corresponds to the minimal difference between the probabilities of reaching a block in a lower layer in a given time among all the blocks in that layer.

Algorithm 1 A Lumping Algorithm

```
1: procedure QLumping(T1, T2)
2:     Assume: T1 & T2 execution tree with states S1 & S2
3:     δ ← 0
4:     n ← 0
5:     P ← ∅                                          ▷ Initial Partition
6:     while n ≤ Height(T1 ⊕ T2) do
7:         L1 ← Layer(T1, n)                          ▷ Current layer in T1
8:         L2 ← Layer(T2, n)                          ▷ Current layer in T2
9:         L ← {L1 ∪ L2}                              ▷ Current layer as single block
10:        Ls ← LowSplit(L)                           ▷ Use low variables for partition
11:        if n ≠ 0 then                              ▷ Except for the leaves
12:            Ls ← ChiSplit(Ls)
13:        end if
14:        P ← P ∪ Ls                                 ▷ Add partitioned layer to P
15:        δ ← max(δ, CompDelta(L1, L2))
16:        n ← n + 1                                  ▷ Go to next level
17:    end while
18:    return δ
19: end procedure
```

The sub-routine LowSplit(L) splits a block L according to the values of the low variables, i.e. it returns a set of blocks $\{B_k\}$ where for any $c_i, c_j \in B_k$ we have $c_i =_L c_j$.

The sub-routine ChiSplit(L) does something similar based on the joint distributions χ_c, i.e. given a set of blocks $\{B_k\}$ it returns a set of blocks $\{B_l\}$ such that the B_ls are sub-blocks of the B_ks (i.e. for every B_l we have $B_l \cap B_k = B_l$ or $B_l \cap B_k = \emptyset$) and for any $c_i, c_j \in B_l$ we have $\chi_{c_i} = \chi_{c_j}$.

In order to compute the best matches for the χ's in T_1 and T_2 on the current level we utilise the sub-procedure CompDelta(L_1, L_2). Note that it returns 1 if the layer does not contain any representative of one of the two trees. This is in fact the maximal distance of two configurations.

Algorithm 2 Algorithm for computing δ

```
1: procedure CompDelta(L1, L2)
2:     β ← 1
3:     while L1 ≠ ∅ do
4:         choose c1 ∈ L1, L1 ← L1 \ c1                ▷ For all c1 ∈ L1
5:         L2' ← L2                                   ▷ New 'working copy' of L2
6:         while L2' ≠ ∅ do
7:             choose c2 ∈ L2', L2' ← L2' \ c2        ▷ For all c2 ∈ L2'
8:             β ← min(β, ‖χc1 − χc2‖∞)               ▷ Find best match
9:         end while
10:        δ ← max(δ, β)
11:    end while
12:    return δ
13: end procedure
```

6.3 Correctness — Relation between ε and δ

The strategy for constructing the lumping described above determines the *coarsest partition* of a set which is stable wrt a given relation [17,18], that is in our case the coarsest PT-bisimulation equivalence. Obviously, this does not necessarily coincide with the 'minimal' one corresponding to the quantity ε defined in [12]. Thus, δ will be in general only a safe approximation, namely an upper bound to the capacity of probabilistic timing covert channel defined by ε. In this section we will show the correctness of our algorithm for computing the estimate δ of the security of a program.

The procedure $\textsc{Layer}(T, n)$ in Algorithm 1 returns the set of configurations whose height in the execution tree T is n. We now define the function *level* which associates to each configuration c in a execution tree T its height in the tree.

Definition 8 Let T be an execution tree for a given program P, and let $c \in \mathbf{Conf}_T$. Then the *level* of c is recursively defined as follows:

$$level(c) = 0 \qquad \text{if } c \text{ is a leaf}$$

$$level(c) = 1 + max\{level(c') \mid c \xRightarrow{p:t} c'\} \qquad \text{otherwise}$$

We will show that the notion of *level* determines a partition which is coarser than the maximum PT-bisimulation (see [18]).

Lemma 1 *Let c_1 and c_2 be two states of a acyclic tPTS T. Then if $c_1 \sim_{PT} c_2$ then $level(c_1) = level(c_2)$.*

Proof By induction on $level(c)$.

The procedure $\textsc{Height}(T)$ in Algorithm 1 returns the height of the execution tree T. This corresponds to the maximum depth reached in a path from the root to a leaf. Thus, we have that if r is the root then $level(r) = \textsc{Height}(T)$.

Proposition 1 *Given two execution trees T_1 and T_2 with roots r_1 and r_2 respectively, then $T_1 \sim_{PT} T_2$ iff $r_1 \sim_{PT} r_2$.*

Proof (if) We show that if $T_1 \not\sim_{PT} T_2$ then $r_1 \not\sim_{PT} r_2$. The hypothesis implies that for all partitions $P \in \mathcal{P}(\mathbf{Conf}_{T_1} \cup \mathbf{Conf}_{T_2})$, if P is a probabilistic time bisimulation then there exists $B \in P$ such that either $B \cap \mathbf{Conf}_{T_1} = \emptyset$ or $B \cap \mathbf{Conf}_{T_2} = \emptyset$. Suppose that $B \in \textsc{Layer}(T_1, i)$, for some $0 \leq i \leq h$, where $h = \max(level(r_1), level(r_2))$, and consider a path ρ from r_1 to a configuration $c \in B$. Then we can show by induction that for all $j > i$ there exists $c_2 \in \textsc{Layer}(T_2, j)$ and $c_1 \in \rho \cap \textsc{Layer}(T_1, j)$, such that for some $B' \in P$ and t, $\chi_{c_1}(t, B') > 0$ while $\chi_{c_2}(t, B') = 0$. That means that any mismatch in layer i propagates to all layers j above. In particular, at level h we have that $r_1 \not\sim_{PT} r_2$.

(only if) Let P be the partition on the union tree. By Definition 5 every block $B \in P$ must contain a representative of each tree. Moreover, by Lemma 1 if $c_1, c_2 \in B$ then $level(c_1) = level(c_2)$. Thus, the block at the root level can only contain r_1 and r_2, and by Definition 5 it must contain both.

In Figure 1 we illustrate the behaviour of the algorithm $\textsc{QLumping}(T_1, T_2)$ before stating formally its correctness, that is that the algorithm always returns zero if $T_1 \sim_{PT} T_2$, and vice versa.

This sketch depicts the situation at the i-th iteration of Algorithm 1 where we have to partition layer L_i. At this point all layers below L_i have already been partitioned; moreover their partitions are stable and will not change any further. The configurations c_j in layer L_i can make transition to any layer below. In our example we assume that there are reachable configurations d_k in layer L_{i-1} and configurations e_l in layer L_{i-2}. We indicate the low-equivalence of two configurations by shading the corresponding nodes. We assume only two different environments, namely the one where $l = 0$ (white nodes) and the one where $l = 1$ (grey nodes).

The first observation we make is that $c_1 =_L c_2$, while c_3 has a different value for the low variable. Therefore, $\textsc{LowSplit}(L_i)$ will put c_1 and c_2 in one block, while c_3 will end up in another block. We next concentrate only on c_1 and c_2 and determine their joint distributions χ_{c_k}, that is the distributions on time and configurations, $\mathbf{Dist}(\mathbb{T} \times \mathbf{Conf})$, χ_{c_1} and χ_{c_2} given below.

χ_{c_1}	d_1	d_2	d_3	d_4	d_5	d_6	e_1	e_2
t_1	p_1	0	0	0	0	0	0	0
t_2	0	p_2	0	0	0	0	0	0
t_3	0	0	p_3	0	0	0	0	0
t_4	0	0	0	0	0	0	0	0
t_5	0	0	0	0	0	0	0	0
t_6	0	0	0	0	0	0	0	0
t'_1	0	0	0	0	0	0	p'_1	0
t'_2	0	0	0	0	0	0	0	0

χ_{c_2}	d_1	d_2	d_3	d_4	d_5	d_6	e_1	e_2
t_1	0	0	0	0	0	0	0	0
t_2	0	0	0	0	0	0	0	0
t_3	0	0	0	0	0	0	0	0
t_4	0	0	0	p_4	0	0	0	0
t_5	0	0	0	0	p_5	0	0	0
t_6	0	0	0	0	0	p_6	0	0
t'_1	0	0	0	0	0	0	0	0
t'_2	0	0	0	0	0	0	0	p'_2

To simplify the presentation, let us assume that all transitions to layer L_i and L_{i-1} take the same time,

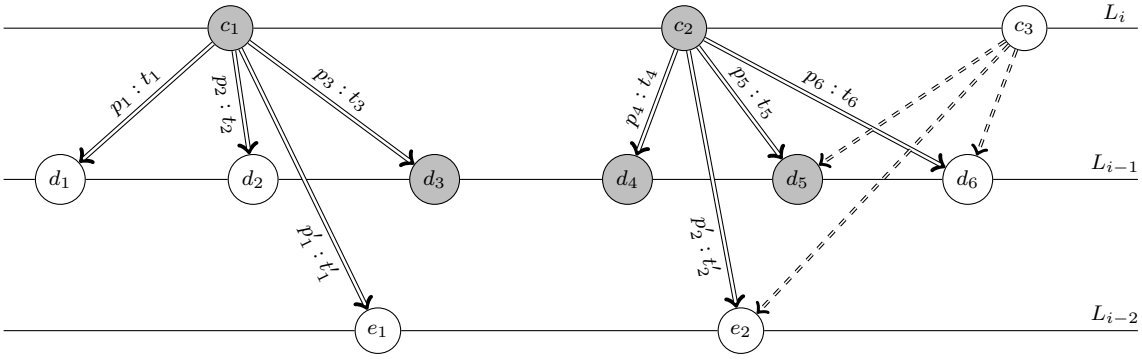


Fig. 1 QLumping at layer L_i

e.g. $t_{1\ldots6} = t$ and $t'_{1,2} = t'$. This is, of course, in general not the case: It is usually possible that configurations in the same layer are reached with different times. This assumption however simplifies the joint distributions we need to consider which becomes:

χ_{c_1}	d_1	d_2	d_3	d_4	d_5	d_6	e_1	e_2
t	p_1	p_2	p_3	0	0	0	0	0
t'	0	0	0	0	0	0	p'_1	0

χ_{c_2}	d_1	d_2	d_3	d_4	d_5	d_6	e_1	e_2
t	0	0	0	p_4	p_5	p_6	0	0
t'	0	0	0	0	0	0	0	p'_2

Note that the normalisation condition for tPTS's means that $p_1 + p_2 + p_3 + p'_1 = 1$ and $p_4 + p_5 + p_6 + p'_2 = 1$.

As said before, the layers below L_i have already been partitioned and we know to which block every configuration belongs to. Clearly, this partition must reflect (i) the layer structure, i.e. the d_k's cannot belong to the same block or class as the e_l's (cf. Lemma 1), and (ii) the value of the low variables forces the differently shaded d_k's into different blocks. For the purpose of this example, let us assume that the blocks in L_{i-1} and L_{i-2} are such that they obey only these two constraints, i.e. that we have three relevant blocks below L_i, namely $B_0 = \{d_1, d_2, d_6\}$ and $B_1 = \{d_3, d_4, d_5\}$ in layer L_{i-1} and $B' = \{e_1, e_2\}$ in layer L_{i-2}. There might be others but they are not reachable from our configurations c_1 and c_2 and we thus can ignore them now. The joint distributions which give the probabilities of reaching blocks in certain times, i.e. $\chi_{c_j} \in \mathbf{Dist}(\mathbb{T} \times \mathcal{P}(\mathbf{Conf}))$ are the following:

χ_{c_1}	B_0	B_1	B'
t	$p_1 + p_2$	p_3	0
t'	0	0	p'_1

χ_{c_2}	B_0	B_1	B'
t	p_6	$p_4 + p_5$	0
t'	0	0	p'_2

Thus, c_1 and c_2 will belong to the same class if and only if $\chi_{c_1} = \chi_{c_2}$, i.e. if and only if $p_1 + p_2 = p_6$, $p_4 + p_5 = p_2$ and $p'_1 = p'_2$.

Proposition 2 *$T_1 \sim_{PT} T_2$ iff* QLumping(T_1, T_2) *returns $\delta = 0$.*

Proof (if) If QLumping(T_1, T_2) returns $\delta = 0$, then for each layer n of T_1 and T_2 the computation of δ must result in zero. Thus, we have that at level $h = \max(\text{Height}(T_1), \text{Height}(T_2))$ the difference $\|\chi_{c_1} - \chi_{c_2}\|_\infty)$ between the two roots must be zero. This implies that both c_1 and c_2 must belong to layer n. In fact, if c_1 would be missing then the above difference would be 1 and similarly for if c_2 would not belong to layer n. Therefore, we have that $c_1 \sim_{PT} c_2$ and by Proposition 1 we conclude that $T_1 \sim_{PT} T_2$.

(only if) Consider the partition $P = \{B_j\}_j$ on $\mathbf{Conf}_{T_1} \cup \mathbf{Conf}_{T_2}$. Lemma 1 guarantees that every block B_j is all contained in one only layer L_i. Moreover, by Definition 5 we have that for all j, $B_j \cap \mathbf{Conf}_{T_1} \neq \emptyset$ and $B_j \cap \mathbf{Conf}_{T_2} \neq \emptyset$.

Suppose by contradiction that QLumping(T_1, T_2) returns $\delta \neq 0$. Then there must be a layer n, $0 \leq n < h$ where Algorithm 2 computes $\delta > 0$. Note that because of Proposition 1 n cannot be h. Let n be the minimal layer where δ becomes not zero. Then the procedure CompDelta() called at the n-th iteration must calculate $\beta > 0$. This means that for all $c_1 \in \text{Layer}(T_1, n)$ and $c_2 \in \text{Layer}(T_2, n)$, $\chi_{c_1} \neq \chi_{c_2}$ must hold. This implies that at least one block at layer n doesn't have a representative in one of the two trees, which contradicts the hypothesis.

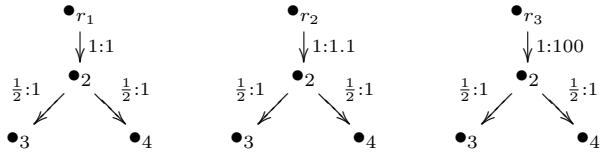
Corollary 1 (Correctness of Algorithm 1) *P is PT-secure iff for any pair of initial configurations c_1, c_2 the corresponding execution trees T_1 and T_2 are such that* QLumping(T_1, T_2) *returns $\delta = 0$.*

Proof By Definition 6 and Proposition 2.

6.4 A Weighted Version: δ'

The actual value of δ is determined by the way we compute the best match between the joint probability distributions χ_{c_1} and χ_{c_2} in line 8 of CompDelta(L_1, L_2). In order to compute δ we use the *supremum norm*, $\|\cdot\|_\infty$, between two distributions, i.e. the largest absolute difference between corresponding entries in χ_{c_1} and χ_{c_2}, respectively. In other words, we try to identify a class of states C (in the layer below) and a time interval t such that the probability of reaching this class in that time from c_1 differs maximally from the one for c_2.

Example 4 To illustrate the possible problems with δ let us consider three very simple PTS's.



It is clear that all three execution trees fail to be bisimilar. If we compute δ via QLumping$(.,.)$ we obtain in the first two layers (starting from the leaves) perfect correspondence: The nodes 3 and 4 in all three trees represent leaf nodes and thus all belong to the same class. The same is true for the states 2 in all three trees – they can all only go to nodes in the leaf node class with the same accumulated probability 1, thus they also belong to the same class, which we can call C_2. This means that up to this level the (maximal) δ we obtain is still 0.

If it comes to the third iteration however, we can then finally establish that the three trees are not bisimilar and that $\delta \neq 0$. We have three different time stamps $t_1 = 1$, $t_2 = 1.1$ and $t_3 = 100$. If we consider the probabilities of reaching the class C_2 in each of the execution trees by starting from the root nodes r_1, r_2 and r_3 then the joint distributions $\chi_{r_i} \in \mathbf{Dist}(\{t_1, t_2, t_3\} \times \{C_2\})$ are represented by 3×1 matrices, as we can only reach a single class, namely C_2. Concretely we have:

χ_{r_1}	C_2
t_1	1
t_2	0
t_3	0

χ_{r_2}	C_2
t_1	0
t_2	1
t_3	0

χ_{r_3}	C_2
t_1	0
t_2	0
t_3	1

If we compute δ_{ij} between tree i and j as the norm difference we get

$$\delta_{ij} = \|\chi_{r_i} - \chi_{r_j}\|_\infty = 1$$

for all $i \neq j$. Note that we would get essentially the same δ using any other vector norm.

One can argue that this is a fair approach as we treat all classes and time labels the same way, namely as completely different. However, it might be useful to develop a measure which reflects the fact that certain times and classes are more similar than others. In the above cases it might seem reasonable to say that the first two trees which have a minimal running difference in the first step are more similar than they are both to the last tree which has an about 100 times longer execution time.

From the point of view of the attacker, such a measure would encode his/her ability in detecting similarity as given by the nature and the precision of the instruments he/she is actually using. For example, suppose it is possible to reach the same class C from c_1 and c_2 with different times t_1 and t_2, such that the corresponding probabilities determine δ (i.e. we have the maximal difference in this case). However, we might in certain circumstances also want to express the fact that t_1 and t_2 are more or less similar, e.g. for $t_1 = 10$ and $t_2 = 10.5$ we might want a smaller δ' than for $t_1 = 1$ and $t_2 = 100$. In terms of the attacker, this means that we make our estimate dependent on the actual power of the time detection instrument that he/she possesses.

In order to incorporate similarity of times and/or classes we need to modify the way we determine the best match in line 8 of CompDelta(L_1, L_2). Instead of determining the norm between χ_{c_1} and χ_{c_2} we can compute a weighted version as:

$$\beta \leftarrow \min(\beta, \|\omega \cdot \chi_{c_1} - \omega \cdot \chi_{c_2}\|_\infty)$$

or simply

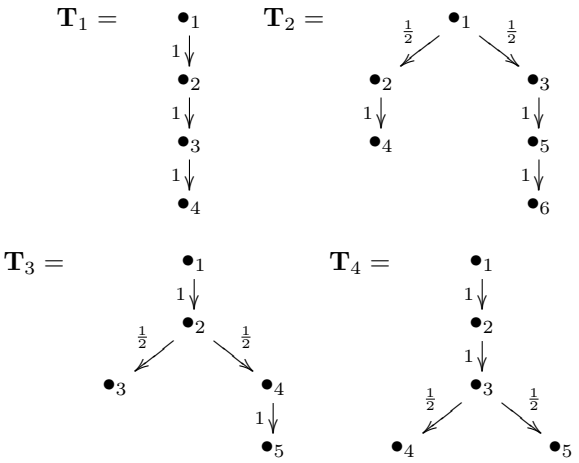
$$\beta \leftarrow \min(\beta, \|\omega \cdot (\chi_{c_1} - \chi_{c_2})\|_\infty),$$

where ω re-scales the entries in χ_{c_1} and χ_{c_2} so as to reflect the relative importance of certain times and/or classes. Note that "$\cdot$" denotes here the component-wise and not the matrix multiplication: $(\omega \cdot \chi)_{tC} = \omega_{tC}\chi_{tC}$. If, for example, an attacker is not able to detect the absolute difference between times but can only measure multiplicities expressing approximative proportions, we could re-scale the χ's via $\omega_{tC} = \log(t)$.

In the following we will use a weighted version δ' which reflects the similarity of classes. The idea is to weight according to the "replaceability" of a class. To this purpose we associate to every class (in the layers below) a matching measure $\mu(C) = \min_{C \neq C'} \delta'(C, C')$, i.e. we determine the δ' between a (sub)tree with a root in the class C in question and all (sub)trees with roots in any of the other classes C'. We can take any representative of the classes C and C' as these are by definition bisimilar. The measure μ indicates how easy it is to replace class C by another one, or how good/precise is

the attacker in distinguishing successor states. Then δ' is simply the weighted version of δ as described above with $\omega_{tC} = \mu(C)$. Note that there is no problem with the fact that δ' is defined recursively as we always know the δ' in the layers below, before we compute δ' in the current layer.

Example 5 In order to illustrate how δ and δ' quantify the difference between various execution trees, let us consider the following four trees.



We abstract from the influence of different transition times and individual state labels, i.e. we assume that $t = 1$ for all transitions and that all states are labelled with the same label.

If we compute the δ and δ' values between all the pairs of systems we get the following results:

δ	$\mathbf{T}_1$	$\mathbf{T}_2$	$\mathbf{T}_3$	$\mathbf{T}_4$
$\mathbf{T}_1$	0.000	0.500	1.000	0.000
$\mathbf{T}_2$	0.500	0.000	1.000	0.500
$\mathbf{T}_3$	1.000	1.000	0.000	1.000
$\mathbf{T}_4$	0.000	0.500	1.000	0.000

δ'	$\mathbf{T}_1$	$\mathbf{T}_2$	$\mathbf{T}_3$	$\mathbf{T}_4$
$\mathbf{T}_1$	0.000	0.250	0.125	0.000
$\mathbf{T}_2$	0.250	0.000	0.125	0.250
$\mathbf{T}_3$	0.125	0.125	0.000	0.125
$\mathbf{T}_4$	0.000	0.250	0.125	0.000

From this we see that δ and δ' are symmetric, i.e. the difference between two systems is symmetric; that every system is bisimilar with itself, i.e. $\delta = 0 = \delta'$ (as we have an empty diagonal); and that the difference between two systems is between zero and one with values in between very well possible.

It is important to note that the theoretical framework, which the measure δ is based on, does not allow for the combination of 'bisimilarity' and 'label similarity' at the base of the definition of δ'. Developing a process algebraic framework supporting a measure like δ' is certainly an interesting route to follow, though not in this paper. As we already pointed out, our aim here is to discuss the analysis of the trade-off between security and other security costs, and how this can be done in a formal way – using essentially the framework of static program analysis – without references to "experience", "experiments" or "good practice".

7 Program Transformation

In [3] Agat introduces a program transformation to remove covert timing channels (*timing leaks*) from programs written in a sequential imperative programming language. The language used is a language of security types with two security levels that is based on earlier work by Volpano and Smith [20,1]. Whilst Volpano and Smith restrict the condition in both while-loops and if-commands to being of the lowest security level, Agat allows the condition in an if-command to be high security providing that an external observer cannot detect which branch was taken. He shows that if a program is typeable in his system, then it is secure against timing attacks. This result depends critically on a notion of Γ- bisimulation: an if-command with a high security condition is only typeable if the two branches are Γ-bisimilar. Agat's notion of bisimilarity is timing aware and based on a notion of low-equivalence which ensures stepwise non-interference. He does not give an algorithm for bisimulation checking.

If a program fails to type, Agat presents a transformation system to remove the timing leak. The transformation pads the branches of if-commands with high security conditions with dummy commands. The objective of the padding is that both branches end up with the same timing and thus become indistinguishable by an external observer. The transformation utilises the concept of a *low-slice*: for a given command C, its low-slice C_L has the same syntactic structure as C but only has assignments to low security variables; all assignments to high security variables and branching on high security conditions are replaced by **SkipAsn** commands of appropriate duration. The transformation involves extending the branches in a high security if-command by adding the low-slice from the other branch. The effect of this transformation is that the timing of the execution of both branches are the same and equal to the sum of timing of the two branches in the non-transformed program. Agat demonstrates that the transformation is semantically sound and that transformed programs are secure.

In order to extend this system to our language, we only have to add a rule for the **choose** statement (essen-

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

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

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

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

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

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

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

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

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

Table 2 Security Typing Rules

tially a straightforward extension of the rule for **if**). In detail, we present the typing rules in Table 2. Note that the rule (If_H) refers to the *semantic* notion of timed bisimilarity, as introduced in Section 5.1.

7.1 Probabilistic Transformation

We consider a probabilistic variant of Agat's language. Probabilities play an important role in the transformation. Rather than just adding the low slice from the other branch to each branch of a high security conditional, we transform each branch to make a probabilistic choice between its padded and untransformed variant. This allows us to trade-off the increased runtime of the padded program versus the vulnerability to attack of the untransformed program. The transformation described is just one on a whole spectrum of probabilistic transformations – at the other extreme we could probabilistically decide whether or not to execute each command in the low slice.

All the formal transformation rules for probabilistic padding are the same as in [3]. The only exception is the rule (If_H): Here we replace – provided certain typing conditions are fulfilled – the branches of an **if** statement not just by the correctly "padded" version as in [3]; 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 code fragment will be executed.

The judgments or transformation rules in Table 3 are of the general form:

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

which represents the fact that with a certain (security) typing Γ we can transform the statement C into D – we also record, as a side-product, the *low slice* D_L of D.

In order to transform programs into secure versions we need to introduce an auxiliary notion, namely the notion of global effect $ge(C)$ of commands. This is used to identify (global) variables which might be changed when a command C is executed. Here is its formal definition (following [3]):

$$\begin{aligned}
ge(x := e) &= \{x\} \\
ge(C_1; C_2) &= ge(C_1) \ \cup\ ge(C_2) \\
ge(\mathbf{if}\ (e)\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2) &= ge(C_1) \ \cup\ ge(C_2) \\
ge(\mathbf{while}\ (e)\ \mathbf{do}\ C) &= ge(C) \\
ge(\mathbf{choose}^p\ C_1\ \mathbf{or}\ C_2) &= ge(C_1) \ \cup\ ge(C_2) \\
ge(\mathbf{skipAsn}\ x\ e) &= \emptyset \\
ge(\mathbf{skipIf}\ e\ C) &= ge(C).
\end{aligned}$$

In our version of the language we have omitted local variables (the **let** statement in Agat's version of the language). As a consequence we have that the side-conditions $ge(D_{1L}) = \emptyset$ and $ge(D_{2l}) = \emptyset$ in rule if_H in Table 3 effectively prevent *any* assignments, i.e. the low slices D_{1L} and D_{2l} are "depleted": they have no effect on the value of any variable but only spend computational time.

Note that our transformation – contrary to the one presented in [3] – is "probabilistic" in the sense that even if we start with a deterministic program C the transformed program D will in general be probabilistic. This is due to rule $(\text{If})_H$ where we introduce a probabilistic choice which depends on a probabilistic parameter $p \in [0, 1]$. To indicate the dependency on p we will also write

$$\Gamma \vdash C \hookrightarrow_p D \mid D_L.$$

In rule if_H we consider the case that a conditional is controlled by a high variable or, more general, a high expression e. In this case we need to obfuscate the timing behaviour. Otherwise it would in general be possible for an attacker who can observe the execution time to distinguish which of the two branches was executed and thus conclude (at least partially) which values the high variables in e had.

The idea in [3] was to add to each branch the "depleted" version of the other branch. This does not change the semantics with respect to what is computed but

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

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

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

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

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

$$(\text{While})\ \frac{\Gamma \vdash_{\leq} e : \mathbf{Bool}_L \quad \Gamma \vdash C \hookrightarrow D \mid D_L}{\Gamma \vdash \mathbf{while}\ (e)\ \mathbf{do}\ C \hookrightarrow \mathbf{while}\ (e)\ \mathbf{do}\ D \mid \mathbf{while}\ (e)\ \mathbf{do}\ D_L}$$

$$(\text{Choose})\ \frac{\Gamma \vdash C_1 \hookrightarrow D_1 \mid D_{1L} \quad \Gamma \vdash C_2 \hookrightarrow D_2 \mid D_{2L}}{\Gamma \vdash \mathbf{choose}^p\ C_1\ \mathbf{or}\ C_2 \hookrightarrow \mathbf{choose}^p\ D_1\ \mathbf{or}\ D_2 \mid \mathbf{choose}^p\ D_{1L}\ \mathbf{or}\ D_{2L}}$$

$$(\text{SkipAsn})\ \frac{}{\Gamma \vdash \mathbf{skipAsn}\ x\ e \hookrightarrow \mathbf{skipAsn}\ x\ e \mid \mathbf{skipAsn}\ x\ e}$$

$$(\text{SkipIf})\ \frac{\Gamma \vdash C \hookrightarrow D \mid D_L}{\Gamma \vdash \mathbf{skipIf}\ e\ C \hookrightarrow \mathbf{skipIf}\ e\ D \mid \mathbf{skipIf}\ e\ D_L}$$

Table 3 Probabilistic Program Transformation

gives both branches the same timing behaviour. Independently of the value of e we thus always get the same time behaviour of the padded conditional. The corresponding rule in [3, fig. 9] is thus:

$$(\text{If}_H)\ \frac{\begin{array}{l}\Gamma \vdash_{\leq} e : \mathbf{Bool}_H\\ \Gamma \vdash C_1 \hookrightarrow D_1 \mid D_{1L} \quad \Gamma \vdash C_2 \hookrightarrow D_2 \mid D_{2L}\\ ge(D_{1L}) = \emptyset \quad ge(D_{2L}) = \emptyset\end{array}}{\begin{array}{l}\Gamma \vdash \mathbf{if}\ (e)\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \hookrightarrow\\ \qquad \mathbf{if}\ (e)\ \mathbf{then}\ (D_1; D_{2L})\ \mathbf{else}\ (D_{1L}; D_2)\\ \qquad \mid \mathbf{skipIf}\ e\ (D_{1L}; D_{2L})\end{array}}$$

Our variation on this theme is to introduce this fixing or padding only with a certain probability p, we thus have after the transformation a probabilistic program which performs with probability p the padded version and with $1 - p$ the original program. The low slice on the right hand side in if_H implements the pure timing behaviour of the padded version of the conditional. We fix the timing leak only with probability p and thus the low slice of the conditional also retains (with this probability) a dependency on the value of e. We could also use a slightly different version of rule if_H which performs the same transformation but describes the low slice slightly differently.

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

7.2 Deterministic Programs

Our next aim is to show that the probabilistic program transformation in Table 3 is correct, in the sense that if we take $p = 1$ in rule $(\text{If})_H$, then the transformation $\Gamma \vdash C \hookrightarrow D \mid D_L$ produces a PT-secure program D. We will approach this issue by first considering the original transformation of deterministic programs by Agat.

The first step we have to consider concerns the relationship between the semantics of $\mathbf{choose}^1\ C_1\ \mathbf{or}\ C_2$ and C_1, and $\mathbf{choose}^0\ C_1\ \mathbf{or}\ C_2$ and C_2, respectively. We use the following notation to denote timing labels:

$$ts ::= t_i \cdot ts \mid \varepsilon$$
$$as ::= t_i \cdot as \mid \surd \cdot as \mid \varepsilon.$$

Execution paths in Agat's language are thus labelled with ts or as depending on whether an execution sequence terminates or not. In the probabilistic language, executions are labelled instead by $p : ts$ or $p : as$, where p denotes the probability that the corresponding path is actually taken. This probability is obtained in the usual way by the product of probabilities along the path.

We will identify deterministic paths labelled by $1 : as$ or $1 : ts$ with paths labelled just with as or ts, respectively. Furthermore, we treat paths with probability zero as 'impossible', i.e. they are not in the transition relation[1], and we identify configurations which can be reached 'instantaneously' i.e. where $as = 0$ or $ts = 0$.

We then can easily establish that the semantics of $\mathbf{choose}^1\ C_1\ \mathbf{or}\ C_2$ and $\mathbf{choose}^0\ C_1\ \mathbf{or}\ C_2$ is identical with that of C_1 and C_2 respectively (provided the choice is made instantaneously).

Lemma 2 *Let C_1 and C_2 be two deterministic programs and assume that $t_{ch} = 0$. Then we have for all environments E that C_1 and* $\mathbf{choose}^1\ C_1\ \mathbf{or}\ C_2$ *are* **semantically equivalent**, *i.e we have:*

$$\langle E \mid C_1\rangle \xrightarrow{ts} \langle E' \mid C_1'\rangle$$

iff

$$\langle E \mid \mathbf{choose}^1\ C_1\ \mathbf{or}\ C_2\rangle \xrightarrow{1:ts} \langle E' \mid C_1'\rangle$$

and

$$\langle E \mid C_1\rangle \xrightarrow{as} E'$$

iff

$$\langle E \mid \mathbf{choose}^1\ C_1\ \mathbf{or}\ C_2\rangle \xrightarrow{1:as} E',$$

and C_2 and $\mathbf{choose}^0\ C_1\ \mathbf{or}\ C_2$ *are* **semantically equivalent**, *i.e*

$$\langle E \mid C_2\rangle \xrightarrow{ts} \langle E' \mid C_1'\rangle$$

iff

$$\langle E \mid \mathbf{choose}^0\ C_1\ \mathbf{or}\ C_2\rangle \xrightarrow{1:ts} \langle E' \mid C_2'\rangle$$

and

$$\langle E \mid C_2\rangle \xrightarrow{as} \langle E' \mid C_1'\rangle$$

iff

$$\langle E \mid \mathbf{choose}^0\ C_1\ \mathbf{or}\ C_2\rangle \xrightarrow{1:as} \langle E' \mid C_2'\rangle.$$

Proof This follows directly from comparing the rules in Table 1 and the ones in Figure 5 in [3].

[1] Arguably this introduces conceptual and/or philosophical problems, but has no influence on the mathematical description of finite computations.

For $t_{ch} \neq 0$ we have essentially the same result except that in this case every execution path starting with $\langle E \mid \mathbf{choose}^{1/0}\ C_1\ \mathbf{or}\ C_2\rangle$ is (slightly) longer then the one starting with C_1 or C_2, respectively. We will assume in the following $t_{ch} = 0$ unless otherwise stated. Our results can be generalised easily to the case $t_{ch} \neq 0$.

For deterministic programs C the transformation in [3] and the one in Table 3 agree (modulo semantical equivalence).

Proposition 3 *Given a deterministic programs C and assuming $t_{ch} = 0$ then if*

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

according to [3, Fig. 9] then also

$$\Gamma \vdash C \hookrightarrow_1 D' \mid D_L'$$

following Table 3, and vice versa, so that D and D_L are semantically equivalent to D' and D_L', respectively.

Proof The proof relies on the fact that deterministic programs C in our languages are also programs in Agat's language with the same typing information Γ. The transformations in [3, Fig. 9] – omitting the (Let) and (Output) rules – are identical to the ones in Table 3 with the exception of rule If_H.

By Proposition 3 we have for $p = 1$ semantically equivalent behaviours for D and D' as well as for D_L and D_L'.

The next step is to relate our notion of PT-security to the one by Agat. In [3] security is described via a slightly different notion of bisimilarity, namely so-called Γ-bisimilarity, which is strong enough to cover deterministic programs.

Definition 9 Γ-Bisimilarity $\sim_\Gamma$ is the largest symmetric relation on commands that satisfies: $C_1 \sim_\Gamma C_2$ if for all E_1, E_2 such that $E_1 =_L E_2$ we have

$$\langle E_1 \mid C_1\rangle \xrightarrow{as} \langle E_1' \mid D_1\rangle$$

implies

$$\langle E_2 \mid C_2\rangle \xrightarrow{as} \langle E_2' \mid D_2\rangle$$

and $E_1' =_L E_2$ and $D_1 \sim_\Gamma D_2$, as well as

$$\langle E_1 \mid C_1\rangle \xrightarrow{ts\cdot\surd} E_1'$$

implies

$$\langle E_2 \mid C_2\rangle \xrightarrow{ts\cdot\surd} E_2'$$

and $E_1' =_L E_2'$.

A program C in Agat's language is then called **Γ-secure** iff $C \sim_\Gamma C$ (cf Definition 2 in [3]. This notion can obviously also be applied to deterministic programs in our language.

For deterministic programs we can see easily that Γ-security implies PT-security based on the following result.

Proposition 4 *Given two deterministic programs C_1 and C_2 then $C_1 \sim_\Gamma C_2$ implies $C_1 \sim_{PT} C_2$.*

Proof We only consider deterministic programs. The distributions χ in the definition of PT-bisimilarity thus degenerate to point distributions. In other words, we have only at most one successor to any given configuration and all execution trees degenerate to single paths without any choice points.

Γ-bisimilarity requires that executions starting in bisimilar configurations reach observable successor configurations (i.e. where low variables change their value or where we have termination $\surd$) along paths with identical labels as or ts. PT-bisimilarity on the other hand requires the same execution length as it is based on the abstract semantics, i.e. PT-bisimilarity requires that the sum of labels, as in Definition 3 (iv), are the same. Clearly $as = as'$ and $ts = ts'$ imply that $\sum as = \sum as'$ and $\sum ts = \sum ts'$, respectively.

The reverse of Proposition 4 does not hold as the following example shows.

Example 6 Consider

$C_1 =$ `h:=0; (if true then h:=0)`

and

$C_2 =$ `(if true then h:=0); h:=0`

The relevant execution sequences leading to termination (no intermediate configuration is observable) for Γ-bisimilarity are

$$\langle E \mid C_1\rangle \xrightarrow{t_e\cdot t_x\cdot t_{asn}\cdot t_e\cdot t_{br}\cdot t_e\cdot t_x\cdot t_{asn}\surd} E'$$

$$\langle E \mid C_2\rangle \xrightarrow{t_e\cdot t_{br}\cdot t_e\cdot t_x\cdot t_{asn}\cdot t_e\cdot t_x\cdot t_{asn}\cdot\surd} E'$$

with $as_1 \neq as_2$. On the other hand we have for the abstract semantics in the definition of PT-bisimilarity:

$$\langle E \mid C_1\rangle \xRightarrow{1:t_e+t_x+t_{asn}+t_e+t_{br}\cdot t_e+t_x+t_{asn}} E'$$

$$\langle E \mid C_2\rangle \xRightarrow{1:t_e+t_{br}+t_e+t_x+t_{asn}+t_e+t_x+t_{asn}} E'$$

which have the same labels.

As an immediate consequence of Proposition 4 we have:

Corollary 2 *Given a deterministic program C then if C is Γ-secure then it is also PT-secure.*

Agat establishes in Theorem 4 in [3] that the (deterministic) transformation of deterministic programs results in secure programs, cf [3, Thm 4].

Theorem 1 *If $\Gamma \vdash C \hookrightarrow D \mid D_L$ then D is Γ-secure.*

As our transformation with $p = 1$ and Agat's are the identical for deterministic programs – more precisely, give semantically equivalent transformed programs, assuming $t_{ch} = 0$ – we thus have reduced the correctness of our transformation in Table 3 for deterministic programs to that of Agat.

Proposition 5 *Given a deterministic program C, assume that $t_{ch} = 0$. Then the transformed program D we obtain with $p = 1$ via*

$$\Gamma \vdash C \hookrightarrow_1 D \mid D_L$$

is PT-secure.

Proof By Theorem 4 in [3] we know that D is Γ-secure and by Corollary 2 it is therefore also PT-secure.

7.3 Probabilistic Programs

We have now established for *deterministic* programs a close relationship between Agat's notion of Γ-security and our notion of PT-security and thus were able to utilise the results in [3] in order to prove the correctness of our transformation in the case of deterministic programs. We will next consider *probabilistic* programs.

In order to cover also probabilistic programs we first show that the choice between two PT-secure programs results in a PT-secure one, i.e. choices do not leak any timing information.

Lemma 3 *Given any two PT-secure programs C_1 and C_2 then* **choose**^{p} C_1 **or** C_2 *is PT-secure for all $p \in [0, 1]$.*

Proof In order to show that **choose**^{p} C_1 **or** C_2 is PT-secure we have to show that for any two environments E and E' with $E =_L E'$ we have $\langle E \mid \textbf{choose}^p\ C_1\ \textbf{or}\ C_2\rangle \sim \langle E' \mid \textbf{choose}^p\ C_1\ \textbf{or}\ C_2\rangle$.

By Definition 4 this means that we have to show that for $\langle E \mid \textbf{choose}^p\ C_1\ \textbf{or}\ C_2\rangle \Rightarrow \chi$ and any other $\langle E' \mid \textbf{choose}^p\ C_1\ \textbf{or}\ C_2\rangle \Rightarrow \chi'$ the successor distributions χ and χ' fulfil $\chi \sim^*_L \chi'$.

We assume, as before, that $t_{ch} = 0$. We then can consider the successor distributions $\langle E \mid C_1\rangle \Rightarrow \chi_1$ and $\langle E \mid C_2\rangle \Rightarrow \chi_2$ in order to construct $\chi = p\chi_1 + (1-p)\chi_2$ as a linear (convex) combination of χ_1 and χ_2. Similarly,

for $\langle E' \mid C_1\rangle \Rightarrow \chi_1'$ and $\langle E' \mid C_2\rangle \Rightarrow \chi_2'$ we get $\chi' = p\chi_1' + (1-p)\chi_2'$.

We know that C_1 and C_2 are PT-secure which means that $\chi_1 \sim_L^* \chi_2$ and $\chi_1' \sim_L^* \chi_2'$. This implies immediately

$$\chi = p\chi_1 + (1-p)\chi_2 \sim_L^* p\chi_1' + (1-p)\chi_2' = \chi'$$

Note that this proof relies essentially on the fact that the successor distribution of a choice construct **choose**^{p} C_1 **or** C_2 can be constructed as a linear combination of the corresponding distributions associated to the alternatives C_1 and C_2. As a consequence it is easy to generalise our arguments to other probabilistic extensions of deterministic languages; for example, allowing for not just two alternatives but n-ary choices.

Proposition 6 *Given any two (probabilistic) programs C_1 and C_2 which are transformed according to:*

$C_1 \hookrightarrow_1 C_1' \mid C_{1L}$
$C_2 \hookrightarrow_1 C_2' \mid C_{2L}$

then **choose**^{p} C_1' **or** C_2' *is PT-secure.*

Proof The proof is by induction on the nested choices in C_1 and C_2. The base case case is for C_1 and C_2 deterministic, which reduces to the previous Lemma 3.

An an immediate consequence of this is the correctness the program transformation in Table 3.

Corollary 3 (Correctness of Transformation) *If $\Gamma \vdash C \hookrightarrow_1 D \mid D_L$ then D is PT-secure.*

We established the correctness of the probabilistic transformation of probabilistic programs (in the case that $p = 1$) based on the correctness of deterministic transformations of deterministic programs as presented in [3]. The critical steps in our line of argument concerned the relation between Γ-bisimilarity and PT-similarity (Lemma 4) and the lifting of PT-security of deterministic programs to probabilistic ones based on considering the linear combinations of the joint distributions χ_i (Lemma 3). This suggests that our approach could be applied easily also to probabilistic extensions of other languages.

8 Cost Analysis

In a recent article on so-called "Software Bugtraps" in *The Economist* the authors report on some ongoing research at NIST on "Software Assurance Metrics and Tool Evaluation" [19]. They claim that "*The purpose of the research is to get away from the feeling that 'all software has bugs' and say 'it will cost this much money to make software of this kind of quality'*", and conclude: "*Rather than trying to stamp out bugs altogether, in short, the future of 'software that makes software better' may lie in working out where the pesticide can be most cost-effectively applied*".

Our aim is to introduce "cost factors" in a similar way into computer security. Instead of trying to achieve perfect security we will look at the trade-off between costs of security counter measures – such as increased average running time – and the improvement in terms of security, which we can measure via the δ introduced above. Even in simple examples we are able to exhibit interesting effects.

8.1 An Example

Our probabilistic version of Agat's padding algorithm allows us to obtain *partially* fixed programs. Depending on the parameter p with which we introduce empty low slices to obfuscate the timing leaks we can determine the (average) execution time of the fixed program in comparison with the improvement in security.

Agat presents in his paper [3] an example which itself is based on Kocher's study [2] of timing attacks against the RSA algorithm. In order to illustrate our approach we simplify the example slightly: The insecure program `agat` we start with is depicted on the left side in Table 4. The fully padded version Agat's algorithm produces, `fagat`, is on the right hand side of Table 4 (to keep things simple we omit Agat's empty statements like `SkipAsn s s`; as `skip` as well as `s:=s` can be used just to 'spend time' without having any real effect on the store we can use e.g. `s:=s` in place of Agat's `SkipAsn s s`). The program, `pagat`, presented in the middle of Table 4 is the result of *probabilistic padding*: The original program `agat` is transformed in such a way that the compensating statements, i.e. low slices, are executed only with probability p while with probability $q = 1 - p$ the original code is executed. For $p = 0$ we have the same behaviour as the original program `agat` while for $p = 1$ this program behaves in the same way as Agat's fully padded version `fagat`.

In our concrete experiments we used the following assumptions. The variable `i` can take values in $\{1, .., 4\}$ while `k` is a three element array with values in $\{0, 1\}$ – nothing is concretely assumed about `s`. The variables `k`, representing a *secret key*, and `s` have security typing H, while `i` is the only low variable which can be observed by an attacker. We implemented this example using (arbitrary) execution times: $t_{asn} = 3$ (assign time), $t_{br} = 2$ (test/branch time), and $t_{skip} = 1$ (skip time), and $t_{ch} = 0$ (choice time).

The abstract semantics for the `pagat` program – which only records choice points and the moments in

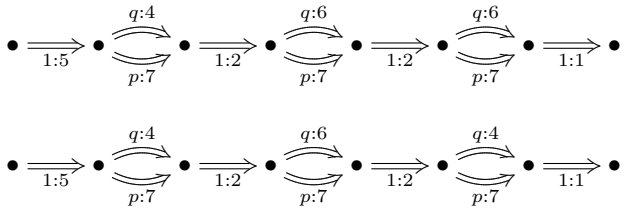
```
i := 1;
while i<=3 do
 if k[i]==1 then
   s := s;
 else
   skip;
 fi;
 i := i+1;
od;
```

```
i := 1;
while i<=3 do
 if k[i]==1 then
   choose p: s := s; skip
   or     q: s := s
   ro
 else
   choose p: s := s; skip
   or     q: skip
   ro
 fi;
 i := i+1;
od;
```

```
i := 1;
while i<=3 do
 if k[i]==1 then
   s := s; skip
 else
   s := s; skip
 fi;
 i := i+1;
od;
```

Table 4 Versions of Agat's Program: `agat`, `pagat`, and `fagat`

time when the low variable changes its value – produces the following execution trees if we start with keys `k=011` and `k=010`:



One can easily see from this how probabilistic padding influences the behaviour of a program: For every bit in the key `k` – i.e. every iteration – we have a choice between executing the original code with probability $q = 1 - p$ or the 'safe' code with probability p. The new code always takes the same time (in our case 7 ticks) while the original code's execution time depends on whether `k[i]` is set or not (either 4 or 6 time steps in our case). Clearly, for $p = 0$ we get in every iteration a different execution time, depending on the bit `k[i]`, and thus can deduce the secret value `k` by just observing the execution times. However, as the execution time is always the same for the replacement code, it is impossible to do the same for $p = 1$. For values of p between 0 and 1, the (average) execution times for `k[i]` $= 0$ and `k[i]` $= 1$ become more and more similar. This means in practical terms that the attacker has to spend more and more time (i.e. repeated observations of the program) in order to determine with high confidence the exact execution time and thus deduce the value of `k[i]` (cf. e.g. [12]).

The price we have to pay for increased security, i.e. indistinguishability of behaviours, is an increased (average) execution time. The graph on the left in Figure 2 shows how the running time (vertical axis) increases in dependence of the padding probability p (horizontal axis) for the eight execution trees we have to consider in this example, i.e. for `k` $=$ `000`, `k` $=$ `001`, `k` $=$ `010`, etc. Depending on the number of bits set in `k` we get four different curves which show how, for example for `k` $=$ `000` the running time increases from 29 time steps (for $p = 0$, i.e. `agat` program) to 38 (for $p = 1$, i.e. `fagat` program).

We can employ the bisimilarity measures δ and δ' in order to determine the security of the partially padded program. For this we compute using our algorithm $\delta(\mathtt{k}_i, \mathtt{k}_j)$ and $\delta'(\mathtt{k}_i, \mathtt{k}_j)$ for all possible keys, i.e. $i, j = 0, \ldots 7$. It turns out that $\delta = 1$ for all values of $p < 1$ and any pair of keys $\mathtt{k}_i$ and $\mathtt{k}_j$ with $i \neq i$; only for $p = 1$ we get, as one would expect, $\delta = 0$ for all key pairs. The weighted measure δ' is more sensitive and we get for example for $p = 0.5$ the following values when we compare $\mathtt{k}_i$ and $\mathtt{k}_j$:

δ'	000	001	010	011	100	101	110	111
000	0.000	0.125	0.250	0.125	0.500	0.125	0.250	0.125
001	0.125	0.000	0.125	0.250	0.125	0.500	0.125	0.250
010	0.250	0.125	0.000	0.125	0.250	0.125	0.500	0.125
011	0.125	0.250	0.125	0.000	0.125	0.250	0.125	0.500
100	0.500	0.125	0.250	0.125	0.000	0.125	0.250	0.125
101	0.125	0.500	0.125	0.250	0.125	0.000	0.125	0.250
110	0.250	0.125	0.500	0.125	0.250	0.125	0.000	0.125
111	0.125	0.250	0.125	0.500	0.125	0.250	0.125	0.000

The diagonal entries are, of course, all zero as every execution tree is bisimilar to itself. The other entries however are different from 0 and 1 and reflect the similarity between the two keys and thus the resulting execution trees. If we plot the development of δ' as a function of p we observe only three patterns as depicted in the right graph in Figure 2. In all three cases δ' decreases from an original value 1 to 0, but in different ways.

In analysing the trade-off between increased running time and security we need to define a *cost* function. For

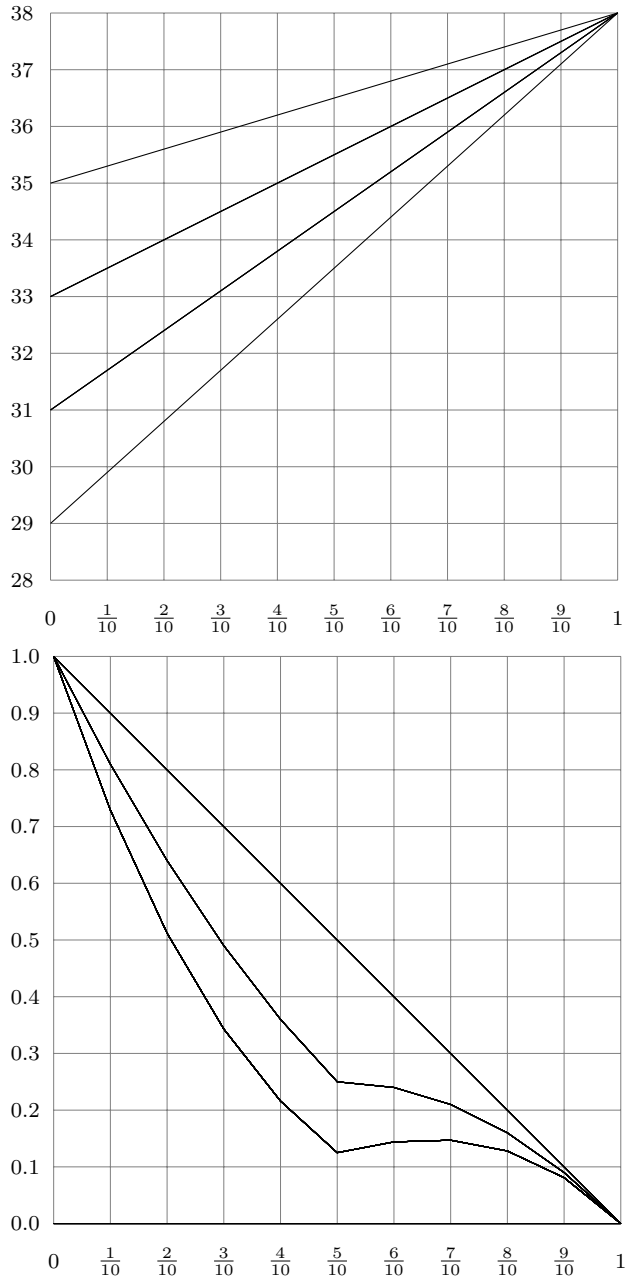


Fig. 2 Running Time $t(p)$ and Security Level $\delta'(p)$ as Functions of p

example, one could be faced with a situation where a certain code fragment needs to be executed in a certain maximal time, i.e. there is a (cost) penalty if the execution takes longer than a certain number of microseconds. In our case we will consider a very simple cost function $c(p) = 6\delta'(p) + t(p)$ with $\delta'(p)$ and $t(p)$ the average δ' between all possible execution trees and t the average running time. The diagram in Figure 3 depicts how $c(p)$, $\delta'(p)$ and $t(p)$ depend on the padding parameter p.

One can argue about the practical relevance of the particular cost function we chose. Nevertheless, this example illustrates already nicely the non-linear nature of security cost optimisation: The optimal, i.e. minimal, cost is reached in this case for $p = 0.5$, i.e. keeping the cost of security counter measures in mind it is better to use a "half-fixed" program rather than a completely safe one.

9 Related and Further Work

The idea of defining a secure system via the requirement that an attacker must be unable to observe different behaviours as a result of different secrets – i.e. the system "operates in the same way" whatever value a secret key has – goes back at least to the seminal work of Goguen and Meseguer [21].

This led in a number of settings to formalisations of security concepts such as "non-interference" via various notions of behavioural equivalences (see e.g. [22–24]). One of the perhaps most prominent of these equivalence notions, namely bisimilarity, plays an important role in the context of the security of concurrent systems. It also finds applications for sequential programs, like e.g. in Agat's work, where bisimulation allows for taking into account the interaction of the programs with the typing environment (see later for more explanations). In order to allow for a decision-theoretic analysis of security countermeasures and associated efforts it appears to be desirable to introduce a "quantitative" notion of the underlying behavioural equivalence. In the case of bisimilarity a first step was the introduction of the notion of *probabilistic bisimulation* by Larson and Skou [10]. However, this notion turns out to be still too strict and a number of researchers developed "approximate" versions. Among them we just name the approaches by Desharnais et.al. [25,26] and van Breugel [27] and our work [13,29] (an extensive bibliography on this issue can be found in [28]). We based this current paper on the latter approach because it allows for an implementation of the semantics of pWhile via linear operators, i.e. matrices, and an efficient computation of δ and δ' using standard software such as `octave` [30]. Another way of quantifying security leakage is via a measurement of the information flow. Various proposals appear in the literature which use nondeterminism [31], belief theory [32], information theory [33–35].

Further research will be needed in order to clarify the relation between our approach and the abovementioned information theoretic approaches.

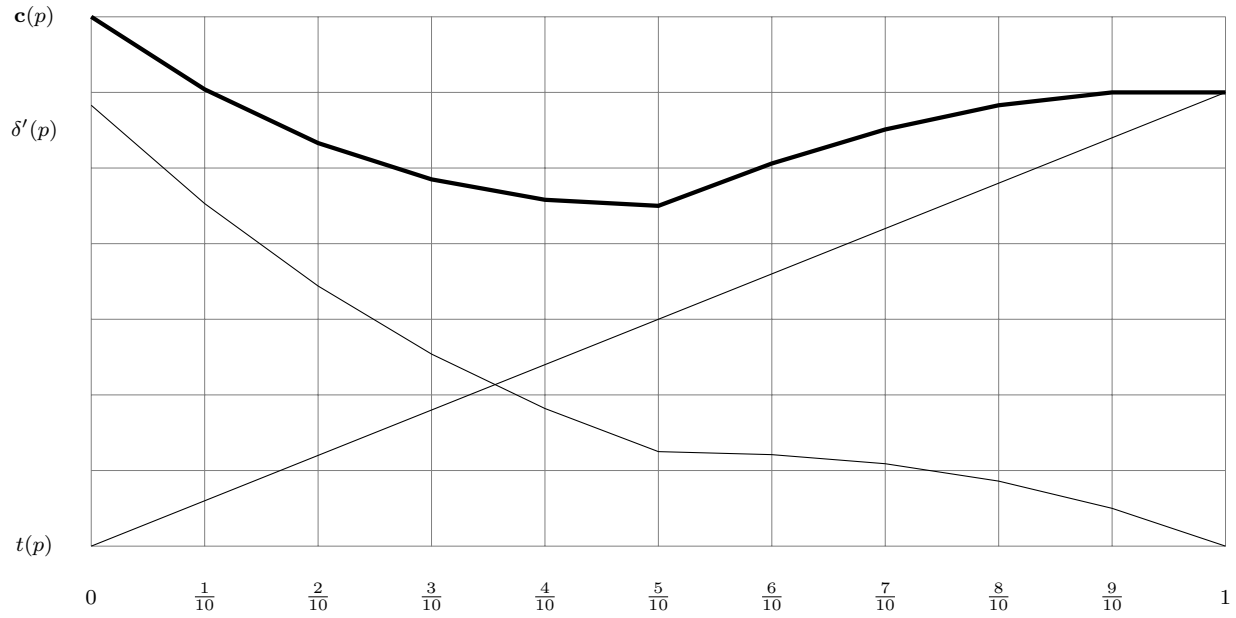


Fig. 3 Trade-Off and Costs $c(p)$ as a Functions of p

10 Conclusions

In this paper we developed a notion of security, PT-security, which lifts the concepts of Γ-bisimilarity and Γ-security introduced in [3] for deterministic programs to probabilistic programs. We discussed in Section 3.2 the possible problems with such a generalisation and pointed out that our lifting is able to model the correlation between the two aspects of time and probability rather than considering them as independent of each other. We then extended the padding based program transformation in [3] in two ways. Firstly, our transformation can be applied to deterministic as well as probabilistic programs, and, secondly, the padding is parametric, i.e. we can introduce the compensating code fragments with a certain, pre-described probability p which expresses the the "strength" or "intensity" of the repair transformation.

We then established the correctness of our probabilistic padding in the case of $p = 1$. Having shown the conservativity of our approach with respect to the deterministic case, in our proofs we could re-use the results for the deterministic case, i.e. we could lift Agat's results for Γ- bisimilarity and Γ-security.

Clearly, padding leads to a performance penalty. We then shown how the probabilistic parameter of our transformation can be utilised for trading security for performance showing a method which involves a cost analysis similar to the ones well known in economics [36,37].

In order to deal with such optimisation problems where we try to balance security and additional security costs, it was necessary to consider not only approximate counter measures (probabilistically fixing leaks) but also approximate notions and quantitative measures of security. A number of such security measures have been proposed within the last decade – e.g. [11] or information theoretic ones like in [33,34,14,35]. In this paper, our aim was not to extend the list or compare the advantages and disadvantages of these proposals. Instead, we were mainly interested in fixing security leaks in a approximate way. Our central theses are: (i) it can make sense to **not** fix a time leak completely but instead – given the additional security costs – to settle for a partial solution; and (ii) the analysis of such a trade-off situation (security vs. costs) can be done in a formal way.

In conclusion, coming back to the question posed in the title of this paper: Should we close a timing covert channel or not? our answer triggers a counter question: How much are you willing to pay for security? As we also propose a continuum between complete closure and non-closure, depending on how much noise we introduce, we are able to achieve a desired level of security in exchange for a certain increase in 'costs', e.g. average running time. The answer to the title question, thus, is a problem of optimal resource allocation: Close the channel as much as you need in order to achieve the best security for the price you are willing to pay. It may seem surprising, but it might be more economical not to close a leak completely.

References

1. Smith, G., Volpano, D.: Secure information flow in a multi-threaded imperative language. In: Proceedings of POPL'98, ACM Press (1998) 355–364
2. 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
3. Agat, J.: Transforming out timing leaks. In: Proceedings of POPL'00, ACM Press (2000) 40–53
4. Mclean, J.: Security models and information flow. In: Proceedings of IEEE Symposium on Security and Privacy (1990) 180–189
5. Di Pierro, A., Hankin, C., Wiklicky, H.: Quantifying timing leaks and cost optimisation. In: L. Chen, M.D. Ryan, G. Wang, eds.: Proceedings of 10th International Conference on Information and Communications Security, Volume 5308 of Lecture Notes in Computer Science., Springer Verlag (2008) 81–96
6. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126**(2) (1994) 183–235
7. Kwiatkowska, M., Norman, G., Sproston, J., Wang, F.: Symbolic model checking for probabilistic timed automata. In Lakhnech, Y., Yovine, S., eds.: Proceedings of FORMATS/FTRTFT'04. Volume 3253 of Lecture Notes in Computer Science., Springer Verlag (2004) 293–308
8. Jonsson, B., Yi, W., Larsen, K. In: Probabilistic Extentions of Process Algebras. Elsevier Science, Amsterdam (2001) 685–710
9. Stirzaker, D.: Probability and Random Variables. Cambridge University Press (1999)
10. Larsen, K., Skou, A.: Bisimulation through probabilistic testing. Information and Computation **94** (1991) 1–28
11. Di Pierro, A., Hankin, C., Wiklicky, H.: Approximate non-interference. In: Proceedings of CSFW'02, IEEE Computer Society (2002) 3–17
12. Di Pierro, A., Hankin, C., Wiklicky, H.: Measuring the confinement of probabilistic systems. Theoretical Computer Science **340**(1) (2005) 3–56
13. 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
14. Smith, G. On the foundations of quantitative information flow. In De Alfero, L., ed.: Proceedings of FOSSACS'09. Volume 5504 of Lecture Notes in Computer Science., Springer Verlag (2009) 288–302
15. Paige, R., Tarjan, R.: Three partition refinement algorithms. SIAM Journal of Computation **16**(6) (1987) 973–989
16. 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
17. Derisavi, S., Hermanns, H., Sanders, W.H.: Optimal state-space lumping in Markov chains. Information Processing Letters **87**(6) (September 2003) 309–315
18. Dovier, A., Piazza, C., Policriti, A.: An efficient algorithm for computing bisimulation equivalence. Theoretical Computer Science **311**(1-3) (2004) 221–256
19. Software Bugtraps: Software that makes software better. Economist **386**(8570) (March 2008)
20. Volpano, D., Smith, G.: Confinement properties for programming languages. SIGACT News **29**(3) (September 1998) 33–42
21. Goguen, J., Meseguer, J.: Security policies and security models. In: Symposium on Security and Privacy, IEEE (1982) 11–20
22. Ryan, P., Schneider, S.: Process algebra and non-interference. Journal of Computer Security **9**(1/2) (2001) 75–103 Special Issue on CSFW-12.
23. Focardi, R., Gorrieri, R.: Classification of security properties (Part I: Information flow). In: Foundations of Security Analysis and Design – Tutorial Lectures. Volume 2171 of Lecture Notes in Computer Science., Springer Verlag (2001) 331–396
24. Aldini, A., Bravetti, M., Di Pierro, A., Gorrieri, R., Hankin, H., Wiklicky, H.: Two formal approaches for approximating noninterference properties. In: Foundations of Security Analysis and Design II – Tutorial Lectures. Volume 2946 of Lecture Notes in Computer Science., Springer Verlag (2002) 1–46
25. Desharnais, J., Jagadeesan, R., Gupta, V., Panangaden, P.: Metrics for labeled Markov systems. In: Proceedings of CONCUR'99. Volume 1664 of Lecture Notes in Computer Science., Springer Verlag (1999) 258–273
26. Desharnais, J., Jagadeesan, R., Gupta, V., Panangaden, P.: The metric analogue of weak bisimulation for probabilistic processes. In: Proceedings of LICS'02, IEEE (2002) 413–422
27. van Breugel, F.: A behavioural pseudometric for metric labelled transition systems. In Abadi, M., de Alfaro, L., eds.: Proceedings of CONCUR'05. Volume 3653 of Lecture Notes in Computer Science., Springer Verlag (2005) 141–155
28. ABE'08: Workshop on Approximate Behavioural Equivalences (2008) `www.cse.yorku.ca/abe08`.
29. Di Pierro, A., Hankin, C., Wiklicky, H.: Approximate non-interference. Journal of Computer Security **12**(1) (2004) 37–81
30. Eaton, J.W.: Octave. Technical report, Free Software Foundation, Boston, MA (2005)
31. Lowe, G.: Quantifying information flow. In: Proceedings of 15th Computer Security Foundations Workshop., IEEE (2002), 18–31
32. Clarkson, M.R., Myers, A.C., Schneider, F.B.: Belief in information flow. In: Proceedings of 18th Computer Security Foundations Workshop., IEEE (2005), 31–45
33. Clark, D., Hunt, S., Malacaria, P.: Quantitative information flow, relations and polymorphic types. Journal of Logic and Computation **15**(2) (2005) 181–199
34. Boreale, M.: Quantifying information leakage in process calculi. In Bugliesi, M., Preneel, B., Sassone, V., Wegener, I., eds.: Proceedings of ICALP'06. Volume 4052 of Lecture Notes in Computer Science., Springer Verlag (2006) 119–131
35. Köpf B., Dürmuth, M.: A provably secure and efficient countermeasure against timing attack. In: Proceedings of 22nd Computer Security Foundations Symposium., IEEE (2009)
36. Hiller F.S., Lieberman G.J.: Introduction to Operations Research. 7th Edition, McGraw-Hill (2001)
37. Aubin, J.-P.: Optima and equilibria – An introduction to nonlinear analysis. Volume 140 of Graduate Texts in Mathematics., Springer Verlag (1993)