Quantifying Timing Leaks and Cost Optimisation

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

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

Abstract. 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 show how to measure the security of a program with respect to this notion via a computable estimate of the timing leakage and use this estimate for cost optimisation.

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]).

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 covers 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 to the program to perform a given command. This is very different from other 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 [4]. 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. The resulting model is essentially a Markov Decision Process (MDP) where rewards are interpreted as time durations. In particular,

L. Chen, M.D. Ryan, and G. Wang (Eds.): ICICS 2008, LNCS 5308, pp. 81–96, 2008.

the presence of non-determinism makes MDP models not very appropriate as a base of our quantitative analysis aiming 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 counter-measures 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. Finally, we present a probabilistic variation of Agat's padding algorithm which we use to illustrate – via an example – a technique for formally analysing the trade-off between security costs and protection.

2 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 by 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 time to execute 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 also 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 [5]) where transition probabilities are defined by a joint distribution of two random variables representing the variable updates and 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. [6]).

2.1 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*,* $\longrightarrow \subseteq S \times \mathbb{T} \times [0,1] \times S$ *a probabilistic transition relation, and* $\lambda : S \to \mathbb{L}$ *a state labelling function.*

We denote by $s_1 \xrightarrow{p:t} s_2$ the fact that $(s_1, p, t, s_2) \in \longrightarrow$ with $s_1, s_2 \in S$, $p \in [0,1]$ and $t \in \mathbb{T}$. In a general tPTS we can have *non-determinism* in the sense that for two states s_1 and s_2 we may have $s_1 \xrightarrow{1:t_1} s_2$ and $s_1 \xrightarrow{1:t_2} s_2$, which would suggest that it is possible to make a transition from s_1 to s_2 in different times (t_1 and t_2) and probability 1, i.e. certainly. In order to eliminate non-determinism we will consider in this paper only tPTS's which are subject to the following conditions: (i) for all $s \in S$ we have $\sum_{(s,p_i,t_j,s_k)\in\longrightarrow} p_i = 1$, and (ii) for all $t \in T$ there is *at most one* tuple $(s_1, t, p, s_2) \in \longrightarrow$.

The first condition means that we consider here 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. a 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 $\pi(s_1, t, p_2) = p$. Note however, that it is still possible to have differently timed transitions between states, i.e. it is possible to have $(s_1, t_1, p_2, s_2) \in \longrightarrow$ and $(s_1, t_2, p_2, s_2) \in \longrightarrow$ with $t_1 \neq t_2$. If 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}.

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* $i = 0, 1, 2, \ldots$.

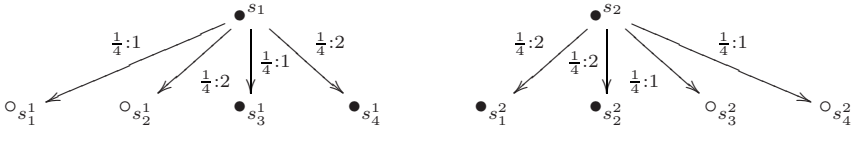
We associate, in the obvious way, to an execution sequence $\sigma = (s_0, s_1, \ldots)$ three more sequences: (i) the transition probability sequence: $(p_1, p_2, \ldots)$, (ii) a time stamp sequence: $(t_1, t_2, \ldots)$, and (iii) a state label sequence: $(\lambda(s_o), \lambda(s_1), \ldots)$.

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 probability distributions can replace measures as they are equivalent in this case. 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.

2.2 Observing tPTS's

In Section 3 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 which 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.

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



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 the $\bullet$ 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 on the rhs 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 the system

on the lhs 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 are $\chi_1(t,l) = \frac{1}{4}$ for all $t = 1, 2$ and $l = \bullet, \circ$; and $\chi_2(1,\circ) = \frac{1}{2} = \chi_2(2,\bullet)$ and $\chi_2(t,l) = 0$ otherwise. 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.

3 An Imperative Language

We consider a language similar to that used in [3] with the addition of a probabilistic choice construct. The syntax of the language, which we call pWhile, is as follows:

$$
\begin{array}{lll}
\text{Operators:} & op & ::= + \mid * \mid - \mid = \mid != \mid < \mid <= \\
\text{Expressions:} & e & ::= v \mid x \mid e\ op\ e \\
\text{Commands:} & C, D & ::= x := e \mid \mathbf{skipAsn}\ x\ e \mid \mathbf{if}\ (e)\ \mathbf{then}\ C\ \mathbf{else}\ D \mid \mathbf{skipIf}\ e\ C \\
 & & \quad \mid \mathbf{while}\ (e)\ \mathbf{do}\ C \mid C; D \mid \mathbf{choose}^p\ C\ \mathbf{or}\ D \\
\text{Basic Values:} & v & ::= n \mid \mathbf{true} \mid \mathbf{false}
\end{array}
$$

The probabilistic choice, $\mathbf{choose}^p\ C\ \mathbf{or}\ D$, is used in an essential way in the program transformation presented later. We also keep the language of types in [3], although in a simplified form (with $\mathrm{L} \leq \mathrm{H}$ and $s \leq s$):

$$
\begin{array}{lll}
\text{Security levels:} & s & ::= \mathrm{L} \mid \mathrm{H} \\
\text{Base types:} & \overline{\tau} & ::= \mathbf{Int} \mid \mathbf{Bool} \\
\text{Security types:} & \tau & ::= \overline{\tau}_s
\end{array}
\qquad \text{and sub-typing:} \quad \frac{s_1 \leq s_2}{\overline{\tau}_{s_1} \leq \overline{\tau}_{s_2}}.
$$

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 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.

The big step semantics of expressions and the small-step semantics of commands are essentially the same as those in [3]. The only difference is the rule for probabilistic choice which we have added to the original semantics. We refer to the full version of this paper [7] for a complete description of this semantics and we only report here on the probabilistic choice rule(s) (Choose):

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

In this rule, t_{ch} indicates the time it takes to execute a choice command. In general, we will use the time labels $t_.$ to 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.

The rule above 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 to be 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 for our pWhile language according to Definition 1. In this tPTS, the state labels are given by the environment, i.e. $\lambda(\langle E \mid C\rangle) = E$.

3.1 Abstract Semantics

According to the notion of security we consider in this paper, an observer or attacker can only observe the changes in low variables. Therefore, we can simplify the semantics by 'collapsing' the execution tree in such a way that execution steps during which the value of all low variables is unchanged are combined into one single step. We call an execution sequence σ *deterministic* if $\pi(\sigma) = 1$, and we call it *low stable* if $\lambda(s_i)|_L = l$ for all $s_i \in \sigma$. 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.

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

(i) *there exists a configuration* $\langle E_1' \mid C_1'\rangle$ *such that* $\langle E_1 \mid C_1\rangle \xrightarrow{p:t} \langle E_1' \mid C_1'\rangle$,

(ii) $\langle E_1' \mid C_1'\rangle \xrightarrow{1:t_1} \ldots \langle E_2' \mid C_2'\rangle \xrightarrow{1:t_n} \langle E_2 \mid C_2\rangle$ *is deterministic,*

(iii) $\langle E_1 \mid C_1\rangle \xrightarrow{p:t} \langle E_1' \mid C_1'\rangle \ldots \xrightarrow{1:t_{n-1}} \langle E_2' \mid C_2'\rangle$ *is maximal low stable,*

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

4 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 2). 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 he/she wants to exploit possible interference through both the probabilistic and the timing behaviour of the system. The notion of bisimulation we introduce here allows us to define a stronger security condition: an attacker must be able to distinguish the probabilities that two programs compute a given

result in a given execution time. This is obviously different from being able to distinguish the probability distributions of the results *and* the running time.

Probabilistic bisimulation was first introduced in [8] and refers to an equivalence on probability distributions over the states of the processes. This latter equivalence is defined as a lifting of the bisimulation relation on the support sets of the distributions, namely the states themselves.

An equivalence relation $\sim \subseteq S \times S$ on S can be lifted to a relation $\sim^* \subseteq \mathbf{Dist}(S) \times \mathbf{Dist}(S)$ between probability distributions on S via (cf [5, Thm 1]): $\mu \sim^* \nu$ iff $\forall [s] \in S/_\sim : \mu([s]) = \nu([s])$. It follows that $\sim^*$ is also an equivalence relation ([5, Thm 3]). For any equivalence relation $\sim$ on the set **Conf** of configurations, we define the associated *low equivalence* relation $\sim_L$ by $c_1 \sim_L c_2$ if $c_1 \sim c_2$ and $c_1 =_L c_2$. Obviously $\sim_L$ is again an equivalence relation. We can lift a low equivalence $\sim_L$ to $(\sim_L)^*$ which we simply denote by $\sim_L^*$.

Definition 4. *Given a security typing Γ, a probabilistic time bisimilarity $\sim$ is the largest symmetric relation on configurations such that whenever $c_1 \sim c_2$, then $c_1 \Longrightarrow \chi_1$ implies that there exists 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 bisimilarity relation in which they are related.

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 $(\sim^*)_L$; in fact, only the former is able to take into account the correlation between time and low variables, while the latter would be a straightforward generalisation of the time bisimulation in [3] which is unable to model such a correlation.

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

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

5 Computing Approximate Bisimulation

The papers [9,10] 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. The quantity 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. We show here how to compute a non-trivial upper bound δ to ε by essentially exploiting the algorithmic solution proposed by Paige and Tarjan [11] for computing bisimulation equivalence. This was already adapted to PTS's in [12], where it was used for constructing a padding algorithm as part of a transformational approach to the timing leaks problem. In this approach the computational paths

of a program are transformed so as to make it perfectly secure by eliminating any possible timing covert channel while preserving its I/O behaviour.

The algorithm we present here is an instantiation of that algorithm where the abstract labels are replaced by the statements in a concrete language (pWhile) and their execution times. Moreover, 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 of the given program.

5.1 Computing δ for PT-Bisimulation

Algorithm CompDelta in Table 1 describes the procedure for computing δ used inside the algorithm QLumping on the lhs of Table 1; this latter constructs a lumping (i.e. a PT-bisimulation equivalence) of two tPTS's T_1 and T_2 with states S_1 and S_2, respectively. QLumping follows the algorithmic paradigm for partition refinement introduced by Paige and Tarjan in [11]. The Paige-Tarjan algorithm constructs a partition of a state space Σ which is *stable* for a given transition relation $\rightarrow$. 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 $S \subseteq \Sigma$ and $pre(X) = \{s \in \Sigma \mid s \rightarrow x \text{ for some } x \in X\}$.

In order to check whether two execution trees T_1 and T_2 in our tPTS model are PT-bisimilar, we apply this refinement technique to the set of states formed by the disjoint union of the states in T_1 and T_2. The strategy of QLumping is as follows: the lumping procedure QLumping(T_1, T_2) works iteratively layer by layer starting from the leaves layer, and splits the blocks in the current partition restricted to the current layer. The procedure CompDelta(L_1, L_2) computes for each two layers L_1 and L_2, the maximal difference $\|\chi(s_1) - \chi(s_2)\|_\infty$ between the probabilities to get from states in $T_1 \cap L_1$ and $T_2 \cap L_1$, respectively, into states of layer L_2. In the original lumping procedure this determines a splitting of the

Table 1. Algorithms QLumping and CompDelta

```
1: procedure QLumping(T1, T2)
2:   δ ← 0, n ← 0 and P ← {S1 ∪ S2}
3:   while n ≤ Height(T1 ⊕ T2) do
4:     S ← {B ∩ CutOff(T1 ⊕ T2, n)) | B ∈ P}
5:     while S ≠ ∅ do
6:       choose B ∈ S, S ← S \ B
7:       P ← Splitting(B, P)
8:     end while
9:     L1 ← Layer(T1, n), L2 ← Layer(T2, n)
10:    CompDelta(L1, L2)
11:    n ← n + 1
12:  end while
13: end procedure
```

```
1: procedure CompDelta(L1, L2)
2:   while L1 ≠ ∅ do
3:     choose s1 ∈ L1, L1 ← L1 \ s1
4:     β ← ∞
5:     L ← L2
6:     while L2 ≠ ∅ do
7:       choose s2 ∈ L, L ← L \ s2
8:       β ← min(β, ‖χ(s1) − χ(s2)‖∞)
9:     end while
10:    δ ← max(δ, β)
11:  end while
12: end procedure
```

states in layer L_1. This value is stored in a variable β and compared with the current value of a variable δ which contains the maximal difference up to that iteration. When the lumping algorithm terminates (that is when we have reached the root of the union tree), one of the following situations will occur: either the roots of T_1 and T_2 belong to the same class in the constructed partition (i.e. T_1 and T_2 are PT-bisimilar) or not. In the latter case δ will contain a maximal difference in the transition probabilities of the two processes which makes them non-bisimilar. This is therefore an estimate of the information leakage of the system. Note that, by construction, δ will be zero in the first case.

The strategy for constructing the lumping described above determines the *coarsest partition* of a set which is stable wrt a given relation, 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 [9]. 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. The following proposition is therefore a corollary of Proposition 45 in [9] stating a similar assertion for ε-bisimulation.

Proposition 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 $\delta = 0$.*

5.2 A Weighted Version: δ'

The actual value of δ is determined by the way we compute the best match between the joint probability distributions $\chi(s_1)$ and $\chi(s_2)$ in line 8 of QLumping. 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 $\chi(s_1)$ and $\chi(s_2)$, 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 s_1 differs maximally from the one for s_2.

One can argue that this is a fair approach as we treat all classes and time labels the same way. However, it might be useful to develop a measure which reflects the fact that certain times and classes are 'more similar' than others.

From the point of view of the attacker, such a measure would encode her/his 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 s_1 and s_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

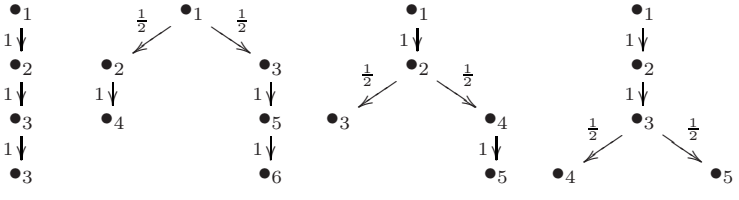
of determining the norm between $\chi(s_1)$ and $\chi(s_2)$ we can compute a weighted version as:

$$\beta \leftarrow \min(\beta, \|\omega \cdot \chi(s_1) - \omega \cdot \chi(s_2)\|_\infty) = \min(\beta, \|\omega \cdot (\chi(s_1) - \chi \cdot (s_2))\|_\infty),$$

where ω re-scales the entries in $\chi(s_1)$ and $\chi(s_2)$ so as to reflect the relative importance of certain times and/or classes. Note that "$\cdot$" denotes here the componentwise 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 2. 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.

6 Cost Analysis

Our aim is to introduce "cost factors" 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 or the weighted δ' introduced above.

6.1 Probabilistic Transformation

In [3] Agat introduces a program transformation to remove covert timing channels (*timing leaks*) from programs written in a sequential imperative programming language. He uses a language of security types with two security levels that is based on earlier work by Volpano and Smith [13,1]. 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 bisimulation; an if-command with a high security condition is only typeable if the two branches are 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 skip 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 untransformed program. Agat demonstrates that the transformation is semantically sound and that transformed programs are secure (correctness).

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 run-time 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 can be found in the full version [7]. The only rule which differs from the original semantics

in [3] is the rule (If_H) given below. 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.

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

6.2 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 2. The fully padded version Agat's algorithm produces, `fagat`, is on the right hand side of Table 2 (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 2 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 dimensional 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 time when the low variable changes its value – produces the following execution trees if we start with keys `k=011` and `k=010`:

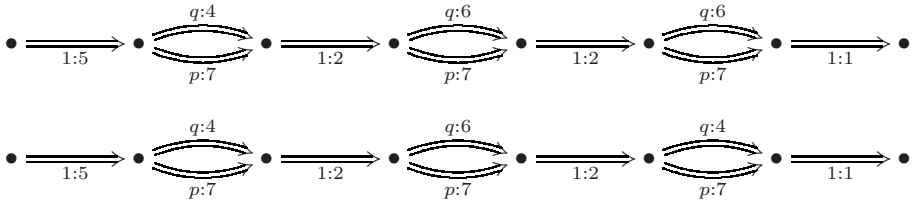


Table 2. Versions of Agat's Program: `agat`,`pagat`, and `fagat`

```
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: skip
   or     q: s := s; 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;
```

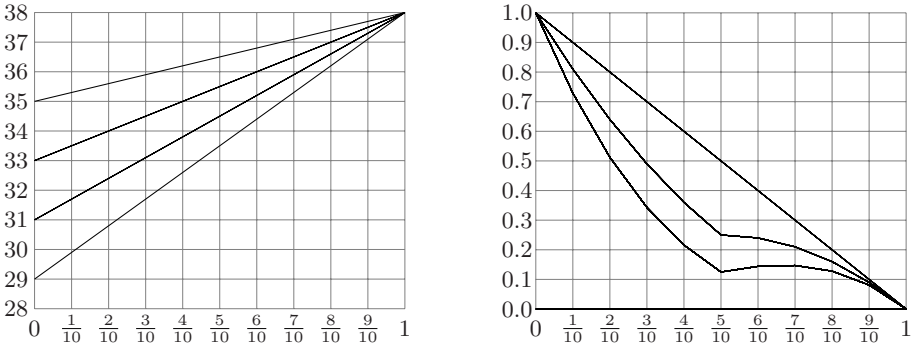


Fig. 1. Running Time $t(p)$ and Security Level $\delta'(p)$ as Functions of p

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 secrete 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. [9]).

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 1 shows how the running time (vertical axis) increases in dependence of

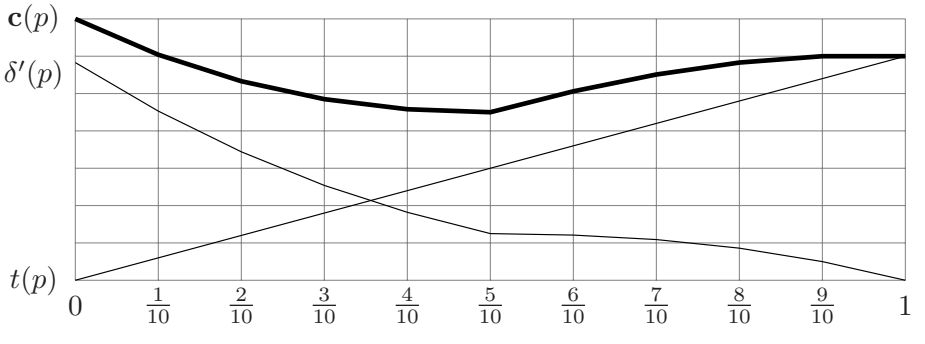
the padding probability p (horizontal axis) for the eight execution trees we have to consider in this example, i.e. for $\mathtt{k} = \mathtt{000}$, $\mathtt{k} = \mathtt{001}$, $\mathtt{k} = \mathtt{010}$, etc. Depending on the number of bits set in $\mathtt{k}$ we get four different curves which show how, for example for $\mathtt{k} = \mathtt{000}$ 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. The $\delta'(\mathtt{k}_i, \mathtt{k}_i)$'s 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. We get for example for $p = 0.5$ the following values for $\delta'(\mathtt{k}_i, \mathtt{k}_i)$:

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

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 1. 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 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 micro-seconds. In our case we will consider a very trivial 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 following diagram depicts how $c(p)$, $\delta'(p)$ and $t(p)$ depend on the padding parameter p.



One can argue about the practical relevance of our particular cost function c. 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 obviously 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.

7 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 work of Goguen and Meseguer [14].

This led in a number of settings to formalisations of security concepts such as "non-interference" via various notions of behavioural equivalencies (see e.g. [15,16]). One of the perhaps most prominent of these equivalence notions, namely *bisimilarity*, plays an important role in the context of security of concurrent systems but also found application for sequential programs such as in Agat's work (as the interaction between system and attacker can be modelled as a parallel composition).

In order to allow for a decision theoretic analysis of security counter-measures 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 [8]. 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. [17,18] and van Breugel [19] and our work [10,20] (an extensive bibliography on this issue can be found in [21]). 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` [22].

Further research will be needed in order to clarify the relation between our measures δ and existing notions of *approximate bisimilarity* mentioned above, e.g. the ε in [9]. Furthermore, we also would like to shed more light on the relationship between our notion and information theoretic concepts used in the work of, for example Clark et.al. [23] and Boreale [24].

References

1. Smith, G., Volpano, D.: Secure information flow in a multi-threaded imperative language. In: POPL 1998, pp. 355–364 (1998)
2. Kocher, P.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996)
3. Agat, J.: Transforming out timing leaks. In: POPL 2000, pp. 40–53 (2000)
4. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
5. Jonsson, B., Yi, W., Larsen, K.: Probabilistic extentions of process algebras. In: Handbook of Process Algebra, pp. 685–710. Elsevier Science, Amsterdam (2001)

6. Stirzaker, D.: Probability and Random Variables. Cambridge University Press, Cambridge (1999)
7. Di Pierro, A., Hankin, C., Wiklicky, H.: Quantifying timing leaks and cost optimisation. Technical Report arXiv:0807.3879 (2008)
8. Larsen, K., Skou, A.: Bisimulation through probabilistic testing. Information and Computation 94, 1–28 (1991)
9. Di Pierro, A., Hankin, C., Wiklicky, H.: Measuring the confinement of probabilistic systems. Theoretical Computer Science 340(1), 3–56 (2005)
10. Di Pierro, A., Hankin, C., Wiklicky, H.: Quantitative relations and approximate process equivalences. In: Amadio, R., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 508–522. Springer, Heidelberg (2003)
11. Paige, R., Tarjan, R.: Three partition refinement algorithms. SIAM Journal of Computation 16(6), 973–989 (1987)
12. Di Pierro, A., Hankin, C., Siveroni, I., Wiklicky, H.: Tempus fugit: How to plug it. Journal of Logic and Algebraic Programming 72(2), 173–190 (2007)
13. Volpano, D., Smith, G.: Confinement properties for programming languages. SIGACT News 29(3), 33–42 (1998)
14. Goguen, J., Meseguer, J.: Security Policies and Security Models. In: IEEE Symposium on Security and Privacy, pp. 11–20 (1982)
15. Ryan, P., Schneider, S.: Process algebra and non-interference. Journal of Computer Security 9(1/2), 75–103 (2001)
16. Focardi, R., Gorrieri, R.: Classification of Security Properties (Part I). In: Focardi, R., Gorrieri, R. (eds.) FOSAD 2000. LNCS, vol. 2171, pp. 331–396. Springer, Heidelberg (2001)
17. Desharnais, J., Jagadeesan, R., Gupta, V., Panangaden, P.: Metrics for labeled markov systems. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 258–273. Springer, Heidelberg (1999)
18. Desharnais, J., Jagadeesan, R., Gupta, V., Panangaden, P.: The metric analogue of weak bisimulation for probabilistic processes. In: LICS 2002, pp. 413–422 (2002)
19. van Breugel, F.: A behavioural pseudometric for metric labelled transition systems. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 141–155. Springer, Heidelberg (2005)
20. Di Pierro, A., Hankin, C., Wiklicky, H.: Approximate Non-Interference. Journal of Computer Security 12(1), 37–81 (2004)
21. ABE 2008: Concur workshop on Approximate Behavioural Equivalences (2008), `www.cse.yorku.ca/abe08`
22. Eaton, J.W.: Octave. Technical report, Free Software Foundation, Boston, MA
23. Clark, D., Hunt, S., Malacaria, P.: Quantitative information flow, relations and polymorphic types. Journal of Logic and Computation 15(2), 181–199 (2005)
24. Boreale, M.: Quantifying information leakage in process calculi. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 119–131. Springer, Heidelberg (2006)