Automatic Covert Channel Analysis of a Multilevel Secure Component

Ruggero Lanotte[1], Andrea Maggiolo-Schettini[2], Simone Tini[1], Angelo Troina[2], Enrico Tronci[3]

[1] Dipartimento di Scienze della Cultura, Politiche e dell'Informazione, Università dell'Insubria
[2] Dipartimento di Informatica, Università di Pisa
[3] Dipartimento di Informatica, Università di Roma "La Sapienza"

Abstract. The *NRL Pump* protocol defines a multilevel secure component whose goal is to minimize leaks of information from high level systems to lower level systems, without degrading average time performances. We define a probabilistic model for the NRL Pump and show how a probabilistic model checker (FHP-murφ) can be used to estimate the capacity of a *probabilistic* covert channel in the NRL Pump. We are able to compute the probability of a security violation as a function of time for various configurations of the system parameters (e.g. buffer sizes, moving average size, etc). Because of the model complexity, our results cannot be obtained using an analytical approach and, because of the low probabilities involved, it can be hard to obtain them using a simulator.

1 Introduction

A computer system may store and process information with a range of classification levels and provide services to users with a range of clearances. The system is *multilevel secure* [2] if users have access only to information classified at or below their clearance. In a distributed framework, multilevel security can be achieved by using trusted secure components, called *Multilevel Secure Components* (MSCs) [8, 9], to connect single-level systems bounded at different security levels, thus creating a *multiple single-level security* architecture [8, 9].

Many applications must satisfy *time performance* as well as *security* constraints which, as well known, are conflicting requirements (e.g. see [20]). This has motivated research into probabilistic protocols that can trade-off between security and time performance requirements. MSCs are an example of such protocols. Their role is to minimize leaks of high level information from high level systems to lower level systems without degrading average time performances.

An MSC proposal is the *Naval Research Laboratory's Pump* (*NRL Pump*) [10–12]. It lets information pass from a low level system to one at a higher level. Now, acks are needed for reliable communication. If the high system passed acks directly to the low one, then it could pass high information by altering ack delays. To minimize such *covert channel*, the pump decouples the acks stream by inserting *random delays.* To avoid time performance degradation, the long-term high-system-to-pump time behavior should be reflected in the long-term pump-to-low-system time behavior. The NRL pump achieves this result by statistically modulating acks: the the pump-to-low-system ack delay is probabilistic based on the moving average of the high-system-to-pump ack delays. This approach guarantees that the average time performances of the secure system (*with* the NRL pump) are the same as those of the insecure one (*without* the NRL pump).

Analysis of *information leakage* of protocols is usually carried out by estimating covert channel capacity [1, 3, 7, 20], which is usually estimated by simulation, since an analytical approach usually cannot handle a full model of the protocol at hand. Using the NRL Pump case study, we show how *probabilistic model checking* can be used to estimate covert channel capacity for various system configurations. This allows a formal as well as automatic security analysis (see [17] for an survey on this subject) of the probabilistic protocol. In particular, we show how FHP-Murφ [4, 5, 22], a probabilistic version of Murφ [6], can be used to compute the probability of security violation of the NRL Pump protocol as function of (discrete) time, for various configurations of the system parameters (e.g. buffer sizes, moving average size, etc). This allows us to estimate the capacity of the probabilistic covert channel left by decoupling the acks stream. Notwithstanding the huge amount of system states, we are able to complete our analysis and to compute covert channel capacity for various NRL pump parameters settings.

Up to our knowledge this is the first time that probabilistic model checking is used for a quantitative analysis of the covert channel capacity. Symbolic model checking, based on PRISM [14–16], has already been widely used for verification of probabilistic protocols. However, for protocols involving arithmetical computations or many FIFO queues, PRISM tends to fill up the RAM [4]. This is due to OBDDs troubles in representing arithmetical operations and FIFO queues. Since probabilistic protocols involving arithmetics or FIFO queues can be often conveniently analyzed using an explicit approach, we use FHP-Murφ to carry out our analysis. Note that indeed the Murφ verifier has already been widely used for security verification, e.g. see [6, 18, 19]. We use FHP-Murφ instead of Murφ since FHP-Murφ extends Murφ capabilities to a probabilistic setting.

We note that an approximate analysis of the covert channel studied here is presented in [13]. However, because of model complexity, only bounds to covert channel capacity can be obtained in [13]. In such cases probabilistic model checking complements the analytical approach by allowing an *exact* analysis on some aspects (i.e., security) of models that are out of reach for the analytical approach.

As for simulation based results (see [12]), the interesting case is when covert channel capacity, as well as the probability of a security violation, is small. Hence, estimating such probabilities by a simulator can be quite hard. In fact, such analysis is not pursued in [12]. In such cases a model checker can be used as an efficient *exhaustive simulator*. Of course a model checker may have to handle a *scaled down* model (with model parameters instanced to *small enough* values) w.r.t. the model handled by a simulator (e.g. w.r.t. the model considered in [12]).

Summing up, we show how probabilistic model checking can complement the covert channel approximate analysis of [13] and the simulation results of [12].

2 The NRL Pump

The NRL Pump is a special purpose trusted device that acts as a router forwarding messages from low level agents to high level ones by monitoring the timing of the acks in the opposite way. As shown in Fig. 1, a low agent sends a message to some high agent through the pump. The pump stores the message in a buffer and sends an ack to the low agent, in order to make communication reliable. The delay of the ack is probabilistically based on a moving average of ack delays from the high agents to the pump. This is an attempt to prevent the high agent to alter the ack timing in order to send information to the low agent. Moreover, the long-term high-agents-to-pump behavior is reflected by the long-term pump-to-low-agents behavior, so that performance is not degraded.

The pump keeps the message in the buffer until the high agent is able to receive it. When the high agent receives the message, it sends an ack to the pump. If the high agent does not acknowledge a received message before a timeout fixed by the pump-administrator expires, the pump stops communication.

$LS \to P$	: $send_L$	low system sends to pump data to deliver to high system
$P \to LS$	: ack_L	pump acknowledges to low system with a probabilistic delay
$P \to HS$	: $send_H$	the pump sends the data to the high system
$HS \to P$	: ack_H	the high system acknowledges to the pump

Fig. 1. Data communication protocol

3 NRL Pump probabilistic ack delay modeling

Let $x_1, \ldots, x_n$ denote the delays of the last n acks sent by the high system to the pump, and $\overline{x}$ denote their average $\sum_{i=1}^{n} x_i/n$. We denote with $p(l, \overline{x})$ the probability that the pump sends an ack to the low system with a delay l. Now, $p(l, \overline{x})$ can be defined in many ways each of which yields different *probabilistic ack schema* with security performances. Let us consider two possible scenarios.

In the first one, $l = \overline{x} + d$, with d a uniformly distributed random variable taking integer values in range $[-\Lambda, +\Lambda]$. Since the expected value of d (notation $E(d)$) is 0, we have $E(l) = \overline{x} + E(d) = \overline{x}$, as required by the NRL Pump specification. We have $p(l, \overline{x}) =$ **if** $(l \in [\overline{x} - \Lambda, \overline{x} + \Lambda])$ **then** $1/(2\Lambda + 1)$ **else** 0.

The drawback of this approach is that, if the schema is known to the low and high systems, then the following *deterministic covert channel* can be established. To transmit bit b ($b = 0, 1$) to the low system, the high system sends h consecutive acks to the pump with a given delay H_b. If h is large enough, from the law of large numbers we will have $\overline{x} \sim H_b$, and $l \in [H_b - \Lambda, H_b + \Lambda]$. W.l.o.g. let us assume $H_0 < H_1$. Whenever the low system *sees* an ack time $l \in [H_0 - \Lambda, H_1 - \Lambda)$ (resp. $l \in (H_0 + \Lambda, H_1 + \Lambda]$), it knows with certainty that a bit 0 (1) has been sent from the high system. Of course, if the ack time is in the interval $[H_1 - \Lambda, H_0 + \Lambda]$ the low system does not know which bit is being sent from the high system. However, if the high system is sending acks with delay H_0 (H_1) and h is large enough, then we know that (with high probability) the low system will observe an ack delay in $[H_0 - \Lambda, H_1 - \Lambda)$ (resp. $(H_0 + \Lambda, H_1 + \Lambda]$). Note that once the low system receives an ack with delay in $[H_0 - \Lambda, H_1 - \Lambda)$ (resp. $(H_0 + \Lambda, H_1 + \Lambda]$) then it is sure that the high system has sent bit 0 (1).

Note that the deterministic covert channel arises since the range of l depends on $\overline{x}$. To solve the problem, in the second scenario we use a *binomial distribution.*

Let $p \in [0, 1]$ and Δ be an integer. Let d be a random variable taking integer values in $[0, \Delta]$ with probabilities: $P(d = k) = \begin{pmatrix} \Delta \\ k \end{pmatrix} p^k (1-p)^{\Delta - k}$. The range of d does not depend on p. Let p be $(\overline{x} - 1)/\Delta$. Since d has a binomial distribution we have $E(d) = \Delta \cdot p = \Delta \cdot \frac{\overline{x}-1}{\Delta} = (\overline{x} - 1)$. We define the pump ack delay l as follows: $l = d+1$. Then, $E(l) = \overline{x}$, as required from the NRL Pump specification.

Since the range of l does not depend on $\overline{x}$, the covert channel of the first scenario does not exist. However the high system can send information to the low one as follows. To transmit bit b $(b = 0, 1)$, the high system sends h consecutive acks to the pump with a given delay H_b. If h is large enough, from the law of large numbers we know that we will have $\overline{x} \sim H_b$. The low system can compute a moving average $\overline{y}$ of the last m ack delays from the NRL Pump. If m is large enough we have $\overline{x} \sim \overline{y}$. Then, by comparing $\overline{y}$ with H_0 and H_1, the low system can estimate (with arbitrarily low error probability) the bit b.

Now, the low system knows the bit b only in a probabilistic sense. In next section we will show that the error probability depends on many parameters, and, by modeling the NRL Pump with FHP-Murφ [4], we will compute such error probability. This, in turn, allows us to estimate the covert channel capacity.

4 FHP-Murφ model of the NRL Pump

FHP-Murφ (*Finite Horizon Probabilistic Murφ*) [4, 5, 22] is a modified version of the Murφ verifier [6, 21]. FHP-Murφ allows us to define *Finite State/Discrete Time Markov Chains* and to automatically verify that the probability of reaching in *at most k* steps a given error state is below a given *threshold.*

Let us describe our FHP-Murφ model of the NRL Pump. We restrict our attention on the NRL Pump features involved in the probabilistic covert channel described above. Moreover, we assume that the low system cannot send any new message until the pump has acknowledged the previous one.

Our goal is to compute the error probability of the low system when it estimates the bit value sent from the high system. This probability depends on the number of consecutive high system acks to the pump with delay H_b and on several parameters, which are defined as constants (see Fig. 2). W.l.o.g. we consider the case $b = 0$. Fig. 2 shows also types used in our model. `BUFFER_SIZE` is the size of the buffer used by the pump to store the messages sent from the low system to the high one. `NRL_WINDOW_SIZE` is the size of the sliding window used by the pump to keep track of the last ack delays from the high system.

```
const   -- constant declarations (i.e. system parameters)
BUFFER_SIZE :    5;   -- size of pump buffer
NRL_WINDOW_SIZE: 5;   -- size of nrl pump sliding window
LS_WINDOW_SIZE:  5;   -- size of low system sliding window
INIT_NRL_AVG :   4.0; -- init. value of high-sys-to-pump moving average
INIT_LS_AVG :    0.0; -- init. value of pump-to-low-syst moving average
DELTA :          10;  -- maximal pump-to-low-system ack delay
HS_DELAY :       4.0; -- H_0 = HS_DELAY; H_1 = HS_DELAY + 2.0
DECISION_THR :   1.0; -- decision threshold
type  -- type definitions (i.e. data structures)
real_type : real(6,99); -- 6 decimal digits, |mantissa| <= 99
BufSizeType : 0 .. BUFFER_SIZE; -- interval
NrlWindow : 0 .. (NRL_WINDOW_SIZE - 1); -- interval
LSWindow : 0 .. (LS_WINDOW_SIZE - 1); -- interval
AckDelay : 0 .. DELTA;  -- range of pump-to-low-system ack delay
```

Fig. 2. Constants (system parameters) and types (data structures)

These delays are used to compute the high-system-to-pump moving average. LS_WINDOW_SIZE is the size of the sliding window used by the low system to keep track of the last ack delays from the pump. These delays are used to compute the pump-to-low-system moving average. INIT_NRL_AVG is the initial value of high-system-to-pump moving average. INIT_LS_AVG is the initial value of pump-to-low-system moving average. DELTA is the maximal pump-to-low-system ack delay: each of these delays ranges in $[0, \texttt{DELTA}]$ and is computed probabilistically. HS_DELAY is such that the delay used by the high system to transmit bit 0 is $H_0 = \texttt{HS\_DELAY}$, and the delay used by the high system to transmit bit 1 is $H_1 = \texttt{HS\_DELAY} + 2.0$. DECISION_THR is such that the low system decides that the bit sent by the high system is b, for $b \in \{0, 1\}$, when it receives an ack from the pump and the difference between the new and the old pump-to-low-system moving average is below DECISION_THR, i.e. when the pump-to-low-system moving average is stable enough. Now, waiting for a long time before making a decision (e.g. by making DECISION_THR small), the low system can be quite sure of making a correct decision, but the more the low system waits for making a decision, the smaller the *bit-rate* of this *covert channel*.

Fig. 3 shows declaration for global variables, which define the state of our model and, hence, the number of bytes needed to represent it (76 in our case). Each variable holds the state of a finite state automaton, whose transitions are defined by a procedure computing the automaton *next state*. The pump model is the synchronous parallel composition of these automata. Variable b represents the number of messages in the pump buffer. Variable nrl_avg is the average of the last delays for the acks received by the pump from the high system. These delays are saved in array nrl_delays, where index nrl_first_index points to the eldest one. Variable ls_avg is the average of the last delays of the acks sent by the pump and received by the low system. These delays are saved in

```
var -- declarations of global variables (i.e. model state variables)
b : BufSizeType;        -- number of msgs in pump buffer
nrl_avg : real_type;     -- high-system-to-pump moving average
ls_avg : real_type;    -- pump-to-low-system moving average
nrl_delays : array[NrlWindow] of real_type;
      -- pump sliding window: last high-system-to-pump ack delays
ls_delays : array[LSWindow] of real_type;
      -- low system sliding window: last pump-to-low-system ack delays
nrl_first_index : NrlWindow;     -- cursor nrl sliding window
ls_first_index : LSWindow; -- cursor low system sliding window
nrl_ack : real_type;   -- pump-to-low-system ack timer
hs_wait : real_type;   -- high-system-to-pump ack timer
ls_decision: 0 .. 1; -- 0: hs sent 0, 1: hs sent 1
ls_decision_state: 0 .. 2; -- 0: to be decided;
     -- 1: decision taken; 2: decision taken and state reset done
```

Fig. 3. Global variables (state variables)

array `ls_delays`, where index `ls_first_index` points to the eldest one. Variable `nrl_ack` is the timer used by the pump for sending the next ack to the low system. Once initialized, at each step `nrl_ack` is decremented by 1. The pump sends the ack when `nrl_ack` ≤ 0. Similarly, `hs_wait` is the timer used by the high system for sending acks to the pump. Variable `ls_decision_state` is 0 if the low system has not yet decided the value of the bit sent by the high system, 1 if the low system has already decided, 2 if the whole system must be restarted after a decision. If `ls_decision_state` is 1 or 2, `ls_decision` takes the value of the bit.

FHP-Murφ allows definition of functions/procedures. Parameters are passed by reference. Function and procedures can modify only those declared “var”.

Procedure `init` (Fig. 4) defines the initial state for our model.

Procedure `nrlpump_ack` (Fig. 5) defines the transitions for the pump-to-low-system ack timer (`nrl_ack`). When it reaches 0, the low system gets an ack from the pump. If the pump buffer is not full, the low system sends a new message, and the pump picks a delay `d` to ack to such message. Delay `d` is an input of the procedure chosen probabilistically by FHP-Murφ rules, as we will see in Fig. 13.

```
procedure init();  begin
 b := 0; nrl_avg := INIT_NRL_AVG;  ls_avg := INIT_LS_AVG;
 for i : NrlWindow do nrl_delays[i] := INIT_NRL_AVG; end;
 for i : LSWindow do ls_delays[i] := INIT_LS_AVG; end;
 nrl_first_index := 0; ls_first_index := 0;  nrl_ack := 0;
 hs_wait := 0;  ls_decision := 0; ls_decision_state := 0;  end;
```

Fig. 4. Procedure `init` defines the initial state

```
procedure nrlpump_ack(Var nrl_ack_new : real_type; d : AckDelay);
begin  -- ack timer expired, low system sends new msg, timer takes d
if ((nrl_ack <= 0.0) & (b < BUFFER_SIZE))
   then nrl_ack_new := d; return; endif;
-- ack timer not expired, waiting ack, timer decremented
if (nrl_ack > 0.0) then nrl_ack_new := nrl_ack - 1; return; endif; end;
```

Fig. 5. Procedure nrlpump_ack defines pump-to-low-system ack timer

```
procedure hs(Var hs_wait_new : real_type);
var last_index : NrlWindow;
begin
-- ack timer not expired, waiting ack, timer decremented
if (hs_wait > 0) then hs_wait_new := hs_wait - 1; return; endif;
-- ack timer expired, high syst receives new msg, timer takes HS_DELAY
if ((hs_wait <= 0.0)&(b > 0)) then hs_wait_new := HS_DELAY;return;endif;
-- ack timer expired, high system waiting for new msg
if ((hs_wait <= 0.0) & (b <= 0)) then hs_wait_new := -2.0; return;endif;
end;  -- hs()
```

Fig. 6. Procedure hs defines high-system-to-pump ack timer

Procedure hs (Fig. 6) defines the transitions for the high-system-to-pump ack timer (hs_wait). The initial value is HS_DELAY since we are in the case $b = 0$.

Procedure buffer (Fig. 7) defines the transition relation for the pump buffer. We have 3 cases: 1) hs_wait and nrl_ack are ≤ 0 (pump received ack from the high system and sent ack to the low one), the pump can send a message to the high system and receive a message from the low one; 2) only hs_wait is ≤ 0 (pump received ack from the high system), the pump can only send; 3) only nrl_ack is ≤ 0 (pump sent an ack to the low system), the pump can only receive.

Procedure nrlpump (Fig. 8) updates the moving average of the high system ack delays. When the pump waits for the ack (hs_wait > 0) it updates the value of the last ack delay. When the hs_wait timer expires ($-1 \leq$ hs_wait ≤ 0), the pump updates the new average for the acks delays and its auxiliary variables.

Procedure obs (Fig. 9) defines the low system estimation of the high system ack delay. As in nrlpump_ack, parameter d is the value of the last ack delay. Initially, the low system updates its information about the last received ack delay as done by the pump (see Fig. 8). If the difference between the new and old pump-to-low-system moving average (fabs(ls_avg - ls_old)) is below DECISION_THR, the pump-to-low-system moving average is stable enough and the low system decides that the high system sent either 0, if HS_DELAY - 1.0 $<$ ls_avg $<$ HS_DELAY + 1.0, or 1, if HS_DELAY + 1.0 $<$ ls_avg $<$ HS_DELAY 3.0. So, the low system may or not take a decision that, in turn, may or not be correct.

```
procedure buffer(Var b_new : BufSizeType);
begin  -- send and get at the same time, b does not change
if (((hs_wait <= 0.0)&(b > 0)) & ((nrl_ack <= 0.0)&(b < BUFFER_SIZE)))
then  return; endif;
-- high system gets msg from buffer
if ((hs_wait <= 0.0)&(b > 0)) then b_new := b - 1; return; endif;
-- low system sends msg to buffer
if ((nrl_ack <= 0.0)&(b < BUFFER_SIZE)) then b_new := b+1;return;endif;
end;
```

Fig. 7. Procedure `buffer` models the pump buffer

```
procedure nrlpump(Var avg_new : real_type); var last_index : NrlWindow;
begin
last_index := (nrl_first_index + NRL_WINDOW_SIZE - 1)%NRL_WINDOW_SIZE;
-- high system processing message
if (hs_wait >  0)
then nrl_delays[last_index] := nrl_delays[last_index] + 1.0; endif;
-- high system sends ack to the pump
if ((hs_wait >=  -1) & (hs_wait <= 0.0)) then
avg_new := nrl_avg +
((nrl_delays[last_index]-nrl_delays[nrl_first_index])/NRL_WINDOW_SIZE);
nrl_first_index := (nrl_first_index + 1)%NRL_WINDOW_SIZE;
nrl_delays[(nrl_first_index+NRL_WINDOW_SIZE-1)%NRL_WINDOW_SIZE] := 0;
endif;  end;
```

Fig. 8. Procedure `nrlpump` updates value of moving average of high syst ack times

Procedure `goto_stop_state` (Fig. 10) resets the NRL Pump (by calling `init`) after the decision has been made by the low system. This procedure has nothing to do with the pump working. It is only used to easy our covert channel measures.

Procedure `main` (Fig. 11) triggers the next state computation of all automata. The parameter `d`, which is passed to `obs` and `nrlpump_ack`, is computed in Fig. 12, where `prob_delay_bin(m,d)` returns the probability that the pump ack time is `d` when the pump moving average is `m`. This function implements a binomial distribution with average value `m` on the interval `[0,DELTA]`.

Fig. 13 shows the definition of the initial state and of the probabilistic transition rules for our model of the NRL Pump.

5 Experimental results

Let us study the probabilities of making a *decision*, the *wrong decision*, or the *right decision* within h time units, denoted $P_{\text{dec}}(h)$, $P_{\text{wrong}}(h)$, and $P_{\text{right}}(h)$ ($= P_{\text{dec}}(h)$ (1-$P_{\text{wrong}}(h)$)). FHP-Murφ returns the probability of reaching a state in which a given invariant fails. Invariant "`no_decision_taken`" in Fig. 14

```
procedure obs(d : AckDelay);
var ls_last_index : LSWindow;     var ackval : real_type;
var ls_old : real_type;           var lsdiff : real_type;
begin
if ((nrl_ack <= 0.0) & (b < BUFFER_SIZE)) then  ackval := d;
 ls_last_index := (ls_first_index + LS_WINDOW_SIZE - 1)%LS_WINDOW_SIZE;
 ls_delays[ls_last_index] := ackval;    ls_old := ls_avg;
 ls_avg := ls_avg + ((ls_delays[ls_last_index] -
                      ls_delays[ls_first_index])/LS_WINDOW_SIZE);
 ls_first_index := (ls_first_index + 1)%LS_WINDOW_SIZE;
 ls_delays[(ls_first_index + LS_WINDOW_SIZE - 1)%LS_WINDOW_SIZE] := 0;
 -- make decision
 if (ls_decision_state = 0) then -- decision has not been taken yet
  -- make decision only when ls_avg stable (i.e. lsdiff small)
  lsdiff := fabs(ls_avg - ls_old);
   if ((lsdiff < DECISION_THR) & (HS_DELAY - 1.0 < ls_avg) &
       (ls_avg < HS_DELAY + 1.0))
       then  ls_decision := 0; ls_decision_state := 1; return; endif;
   if ((lsdiff < DECISION_THR) & (HS_DELAY + 1.0 < ls_avg) &
       (ls_avg < HS_DELAY + 3.0))
     then ls_decision := 1; ls_decision_state := 1; return; endif;
  endif; endif; end;
```

Fig. 9. Procedure `obs` computes the low syst estimate of the high syst ack time

```
-- reset all, but obs_decision_state
procedure goto_stop_state(); begin  init();
ls_decision_state := 2; end;     -- decision taken and reset done
```

Fig. 10. Procedure `goto_stop_state` resets system states

states that no decision has been taken, and allows us to compute $P_{\mathrm{dec}}(h)$. Invariant "`no-dec_or_right-dec`" in Fig. 14 states that no decision or the correct decision has been taken, and allows us to compute $P_{\mathrm{wrong}}(h)$.

We studied how $P_{\mathrm{dec}}(h)$, $P_{\mathrm{wrong}}(h)$ and $P_{\mathrm{right}}(h)$ depend on `BUFFER_SIZE`, `NRL_WINDOW_SIZE`, `LS_WINDOW_SIZE`. Fig. 15 shows $P_{\mathrm{dec}}(h)$ and $P_{\mathrm{wrong}}(h)$, and Fig. 16 shows $P_{\mathrm{right}}(h)$, as functions of h for some parameter settings. We label `A-B-C-d` (resp. `A-B-C-w`, `A-B-C-r`) the experiments referring to $P_{\mathrm{dec}}(h)$ (resp. $P_{\mathrm{wrong}}(h)$, $P_{\mathrm{right}}(h)$) with settings `BUFFER_SIZE = A`, `NRL_WINDOW_SIZE = B`, `LS_WINDOW_SIZE = C`. (Each of the curves in Figs. 15,16 required about 2 days of computation on a 2 GHz Pentium PC with 512 MB of RAM.)

Fig. 15 shows that the low system with probability almost 1 decides correctly the value of the bit sent by the high system within 100 time units. Our time unit is about the time needed to transfer messages from/to the pump. Thus we may reasonably assume that a time unit is about 1ms. Then Fig. 16 tells us that the

```
procedure main(d : AckDelay);
Var b_new : BufSizeType;      Var nrl_ack_new : real_type;
Var hs_wait_new : real_type;  Var avg_new : real_type;
begin  -- decision taken and state reset already done
if (ls_decision_state = 2) then return; endif;
if (ls_decision_state = 0) then -- decision not taken yet
 b_new := b;               nrl_ack_new := nrl_ack;
 hs_wait_new := hs_wait;   avg_new := nrl_avg;
 buffer(b_new);            nrlpump_ack(nrl_ack_new, d);
 obs(d);                   nrlpump(avg_new);
 hs(hs_wait_new);          b := b_new;
 nrl_ack := nrl_ack_new;   hs_wait := hs_wait_new;
 nrl_avg := avg_new;
else -- decision taken but state reset to be done
 goto_stop_state();  endif;  end;
```

Fig. 11. Procedure `main` updates system state

```
function binomial(n : real_type; k : real_type) : real_type;
var result : real_type;      var nn, kk : real_type;
begin   result := 1; nn := n; kk := k;
while (kk >= 1) do
   result := result*(nn/kk);  nn := nn - 1; kk := kk - 1;  endwhile;
return (result);  end;
function prob_delay_bin(m : real_type; d : AckDelay) : real_type;
var p : real_type;
begin   p := m/DELTA;
return ( binomial(DELTA, d)*pow(p, d)*pow(1 - p, DELTA - d) );  end;
```

Fig. 12. Function `prob_delay_bin()` updates high sys ack timer

high system can send bits to the low system at a rate of about 1 bit every 0.1 seconds, i.e. 10 bits/sec. Hence, for many applications the NRL Pump can be considered *secure*, i.e. the covert channel has such a low capacity that it would take too long to the high system to send interesting messages to the low system. On the other hand, it is not hard to conceive scenarios where also a few bits sent from the high system to the low system are a security threat.

Notice that, for the parameter settings we studied, the most important parameter is `LS_WINDOW_SIZE`, i.e., different parameter settings with the same value of `LS_WINDOW_SIZE` yield the same curves in Figs. 15, 16. Moreover, the larger is `LS_WINDOW_SIZE` the later and more precise is the decision of the low system. This is quite natural since `LS_WINDOW_SIZE` is the width of the moving average used by the low system to estimate the average ack delays from the high system. The more samples we use in such a moving average the better is the estimation, event though the longer is the waiting to get it.

```
startstate "initstate" init(); end;  -- define init state
ruleset d : AckDelay do   -- define transition rules
 rule "time step" prob_delay_bin(nrlavg, d) ==> begin main(d) end; end;
```

Fig. 13. Startstate and transition rules for NRL pump model

```
invariant "no_decision_taken" 0.0   (ls_decision_state = 0);
invariant "no-dec_or_right-dec" 0.0
(ls_decision_state = 0) | ((ls_decision_state>0) & (ls_decision=0));
```

Fig. 14. Invariants

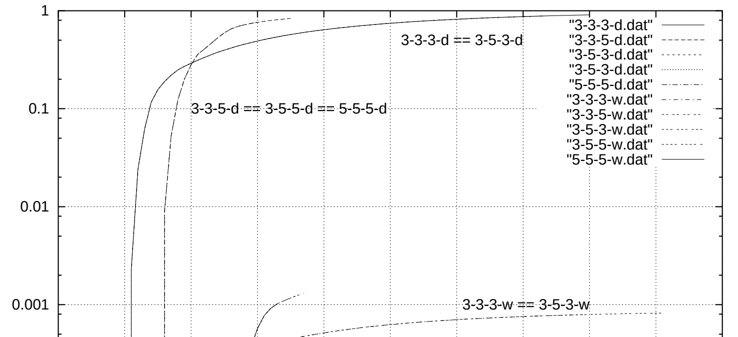

Fig. 15. Probabilities of taking a decision and a wrong decision within h time units

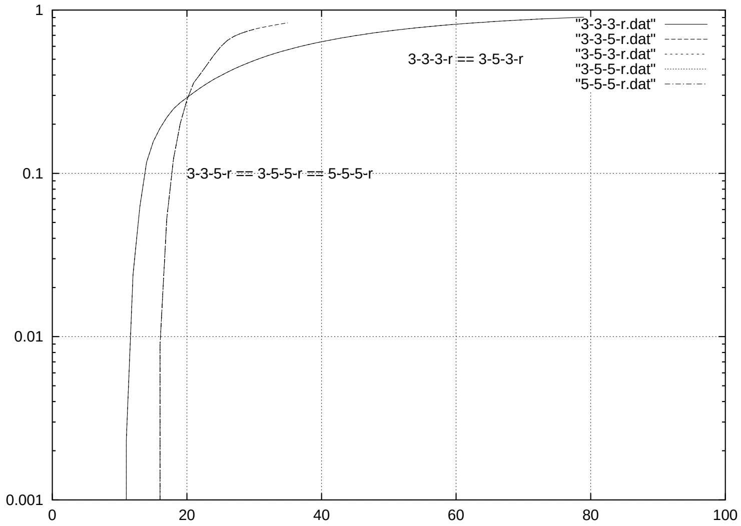

Fig. 16. Probability of taking the right decision within h time units

Fig. 16 shows that the transition from the situation in which no covert channel exists ($P_{\text{right}}(h)$ close to 0) to that in which a covert channel exists ($P_{\text{right}}(h)$ close to 1) is steep. Hence, relatively small changes in system parameters (namely `LS_WINDOW_SIZE`) may have a dramatic effect on security.

6 Conclusions

Using the *NRL Pump* [10–12] as a case study, we show how *probabilistic model checking* can be used to compute covert channel capacity for various system configurations. More specifically, using FHP-Murφ [4], a probabilistic version of the Murφ [6] verifier, we were able to compute the probability of a *security* violation of the *NRL Pump* protocol as function of (discrete) time for various configurations of the system parameters (e.g. buffer sizes, moving average size, etc). This, in turn, allows us to estimate the capacity of the probabilistic covert channel left by decoupling the acks stream. Because of the model complexity, our results cannot be obtained using an analytical approach and, because of the low probabilities involved, can be quite hard to obtain them using a simulator.

Our experiments show that, although model checking has to handle a *scaled down* model (with parameters instanced to *small enough* values) w.r.t. that handled by a simulator, for complex systems probabilistic verification nicely complements a security analysis carried out using analytical and simulation approaches.

A natural next step for our research is to investigate methods to automatically analyze protocols (i.e. Markov Chains) with a larger state space.

References

1. R. K. Abbott and H. Garcia-Molina: Scheduling Real-Time Transactions: a Performance Evaluation. ACM Trans. Database Syst. 17(3), 1992, 513–560.
2. D. Bell and L.J. La Padula: Secure Computer Systems: Unified Exposition and Multics Interpretation. Tech. Rep. ESD-TR-75-301, MITRE MTR-2997, 1976.
3. R. David and S. H. Son: A Secure Two Phase Locking Protocol. Proc. IEEE Symp. on Reliable Distributed Systems, 1993, 126–135.
4. G. Della Penna, B. Intrigila, I. Melatti, E. Tronci, and M. V. Zilli: Finite Horizon Analysis of Markov Chains with the Murphi Verifier. Proc. CHARME, Springer LNCS 2860, 2003, 394–409.
5. G. Della Penna, B. Intrigila, I. Melatti, E. Tronci, and M. V. Zilli: Finite Horizon Analysis of Stochastic Systems with the Murphi Verifier. Proc. ICTCS, Springer LNCS 2841, 2003, 58–71.
6. D. L. Dill, A. J. Drexler, A. J. Hu, and C. Han Yang: Protocol Verification as a Hardware Design Aid. Proc. IEEE Int. Conf. on Computer Design on VLSI in Computer & Processors, 1992, 522–525.
7. J. Gray and A. Reuter: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers Inc., 1992.
8. M. H. Kang, J. Froscher, and I. S. Moskowitz: A Framework for MLS Interoperability. Proc. IEEE High Assurance Systems Engineering Workshop, 1996, 198–205.
9. M. H. Kang, J. Froscher, and I. S. Moskowitz: An Architecture for Multilevel Security Interoperability. Proc. IEEE Computer Security Application Conf., 1997.
10. M. H. Kang, A. P. Moore, and I. S. Moskowitz: Design and Assurance Strategy for the NRL Pump. IEEE Computer 31(4), 1998, 56–64.
11. M. H. Kang and I. S. Moskowitz: A Pump for Rapid, Reliable, Secure Communication. Proc. ACM Conf. on Computer and Communication Security, 1993, 119–129.
12. M. H. Kang, I. S. Moskowitz, and D. Lee: A Network Pump. IEEE Trans. Software Eng. 22(5), 1996, 329–338.
13. I. S. Moskowitz and A. R. Miller: The Channel Capacity of a Certain Noisy Timing Channel. IEEE Trans. Information Theory, 38(4), 1992.
14. PRISM Web Page: http://www.cs.bham.ac.uk/∼dxp/prism/
15. M. Kwiatkowska, G. Norman, and D. Parker: PRISM: Probabilistic Symbolic Model Checker. Proc. TOOLS, Springer LNCS 2324, 2002, 200–204.
16. M. Kwiatkowska, G. Norman, and D. Parker: Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach. Proc. TACAS, Springer LNCS 2280, 2002.
17. C. Meadows: What Makes a Cryptographic Protocol Secure? The Evolution of Requirements Specification in Formal Cryptographic Protocol Analysis. Proc. ESOP, Springer LNCS 2618, 2003, 10–21.
18. J. C. Mitchell, M. Mitchell, and U. Stern: Automated Analysis of Cryptographic Protocols using Murφ. Proc. IEEE Symp. on Security and Privacy, 1997, 141–151.
19. J. C. Mitchell, V. Shmatikov, and U. Stern: Finite-State Analysis of SSL 3.0. Proc. USENIX Security Symp., 1998.
20. S. H. Son, R. Mukkamala, and R. David: Integrating Security and Real-Time Requirements Using Covert Channel Capacity. IEEE Trans. Knowl. Data Eng. 12(6), 2000, 865–879.
21. Murphi Web Page: http://sprout.stanford.edu/dill/murphi.html
22. Cached Murphi Web Page: http://www.dsi.uniroma1.it/∼tronci/cached.murphi.html