On formal and automatic security verification of WSN transport protocols

Ta Vinh Thong[1]
thong@crysys.hu

Amit Dvir[2]
azdvir@gmail.com

Laboratory of Cryptography and System Security (CrySyS Lab.)
Budapest University of Technology and Economics, Hungary[1]

Computer Science Department, The College of Management - Academic Studies, Israel[2]

TECHNICAL REPORT

2013

Contents

Abstract

In this paper, we address the problem of formal and automated security verification of WSN transport protocols that may perform cryptographic operations. The verification of this class of protocols is difficult because they typically consist of complex behavioral characteristics, such as real-time, probabilistic, and cryptographic operations. To solve this problem, we propose a probabilistic timed calculus for cryptographic protocols, and demonstrate how to use this formal language for proving security or vulnerability of protocols. The main advantage of the proposed language is that it supports an expressive syntax and semantics, including bisimilarities that supports real-time, probabilistic, and cryptographic issues at the same time. Hence, it can be used to verify the systems that involve these three properties in a more convenient way. In addition, we propose an automatic verification method, based on the well-known PAT process analysis toolkit, for this class of protocols. For demonstration purposes, we apply the proposed manual and automatic proof methods for verifying the security of DTSN and SDTP, which are two of the recently proposed WSN tranport protocols.

1 Introduction

Numerous transport protocols specifically designed for WSN applications, requiring particularly reliable delivery and congestion control (e.g., multimedia sensor networks) have been proposed [20]. Two of the latest protocols are the Distributed Transport for Sensor Networks (DTSN) [15, 19], and its secured version, the SDTP protocol [5]. In DTSN and SDTP the intermediate nodes can cache the packets with some probability and retransmit them upon request, providing a reliable transmission, energy efficiency and distributed functionality.

Unfortunately, existing transport protocols for WSNs (include DTSN) do not include sufficient security mechanisms or totally ignore the security issue. Hence, many attacks have been found against existing WSN transport protocols [4]. Broadly speaking, these attacks can be classified into two groups: attacks against reliability and energy depleting. Reliability attacks aim to mislead the nodes so that loss of a data packet remains undetected. In the case of energy depleting attacks, the goal of the attacker is to perform energy-intensive operations in order to deplete the nodes' batteries [4]. In particular, using a fake or altered ACK packet, an attacker can give the sender the impression that data packets arrived safely when they may actually have been lost. Similarly, forging or altering $NACK$ packets to trigger unnecessary retransmission can lead to faster draining of the node's batteries. While futile retransmissions do not directly harm the reliability of service, it is still undesirable.

In this paper, we address the problem of formal and automated security verification of WSN transport protocols, which typically consist of the following behavioral characteristics: (b1) storing data packets in the buffer of sensor nodes; (b2) probabilistic and real-time behavior; (b3) performing cryptographic operations such as one-way hashing, digital signature, computing message authentication codes (MACs), and so on.

We propose a formal and an automated verification method, based on the application of a process algebra and a model-checking framework, respectively. For demonstration purposes, we apply the proposed methods for specifying and verifying the security of the DTSN and the SDTP protocols, which are representative in the sense that DTSN involve the first two points of the behavioral characteristics (b1-b2), while SDTP covers all of the three points (b1-b3). Specifically, the main contributions of this paper are the following:

- We propose a probabilistic timed calculus, called $crypt^{prob}_{time}$, for cryptographic protocols. To the best of our knowledge, this is the first of its kind in the sense that it combines the following three features, all at once: (i.) it supports formal syntax and semantics for cryptographic primitives and operations; (ii.) it supports time constructs similar to the concept of timed automata that enables us to verify real time systems; (iii.) it also includes the syntax and semantics of probabilistic constructs for analysing the systems that perform probabilistic behavior. The basic concept of $crypt^{prob}_{time}$ is inspired by the previous works [9, 10, 6] proposing solutions, separately, for each of the three discussed points. In particular,

$crypt^{prob}_{time}$ is a modified and combined version of the well-known concepts of the applied π-calculus [9], which defines an expressive syntax and semantics supporting cryptographic primitives to analyse security protocols; the probabilistic extension of the applied π-calculus [10]; and the process calculus for timed automata proposed in [6].

We note that although in this paper the proposed $crypt^{prob}_{time}$ calculus is used for analysing WSN transport protocols, it is also suitable for reasoning of other systems that include cryptographic operations, real-time and probabilistic behavior.

- Using $crypt^{prob}_{time}$ we specify the behavior of the DTSN and SDTP protocols. We proposed the novel definition of *probabilistic timed bisimilarity* and used it to prove the weaknesses of DTSN and SDTP, as well as the security of SDTP against some attacks.
- We provide the automatic security verification of the DTSN and SDTP protocols with the PAT process analysis toolkit [8], which is a powerful general-purpose model checking framework. To the best of our knowledge PAT has not been used for this purpose before, however, in this paper we show that the power of PAT can be used to check some interesting security properties defined for these systems/protocols.

The structure of the paper is as follows: Due to its complexity, we will introduce $crypt^{prob}_{time}$ in three steps. In Section 2 we start with the introduction of the base calculus, *crypt*, which is a variant of the applied π-calculus [9], designed for analysing security protocols. The extension of *crypt*, called $crypt_{time}$, with real-time modelling elements is given in Section 3, while in Section 4 we provide the description of $crypt^{prob}_{time}$, the probabilistic extension of $crypt_{time}$. The specifications of the DTSN and SDTP protocols in $crypt^{prob}_{time}$ can be found in Section 5 and 6, respectively. The security analysis of DTSN and SDTP, based on $crypt^{prob}_{time}$, is provided in Section 7. The well-known model-checking framework PAT and automatic verification of DTSN and SDTP are described in Section 8. Finally, we conclude the paper and talking about future works in Section 9.

2 *crypt*: The calculus for cryptographic protocols

crypt is the base calculus for specifying and analysing cryptographic protocols, without supporting real-time and probabilistic systems. *crypt* can be seen as a modified variant of the well-known applied π-calculus [9], designed for analysing security protocols, and proving security properties of the protocols in a convenient way. Our goal is to extend *crypt* with time and probabilistic modelling elements *adopting the well-defined concept of timed and probabilistic automata*, and to do this, we need to modify the applied π-calculus in some points.

2.1 Syntax and semantics

We assume an infinite set of *names* $\mathcal{N}$ and *variables* $\mathcal{V}$, where $\mathcal{N} \cap \mathcal{V} = \emptyset$. Further, we define a set of distinguished variables $\mathcal{E}$ that model the cache entries for specifying the systems including entities that store data in their cache entries. In the set $\mathcal{N}$, we distinguish channel names, and other kind of data. We let the channel names range over c_i with different indices such that $c_i \neq c_j$, $i \neq j$. The set of non-negative integers is denoted by $\mathcal{I}$, and its elements range over int_i with different indices that are corresponding to the numbers 0, 1, 2, etc.

Further, we let the remaining data names range over m_i, n_i, k_i. The variables range over x_i, y_i, z_i, and the cache entries range over e_i with different indices. The names and variables with different indices are different. We let $\sum$ be the set of function symbols. To verify security protocols, in our case the function symbols capture the cryptographic primitives such as hash, digital signature, encryption, MAC function. Finally, we assume the well-defined type system of the terms as in the applied π-calculus.

We define a set of terms as

$$t ::= c_i \mid int_i \mid n_i, m_i, k_i \mid x_i, y_i, z_i \mid e_i \mid f(t_1, \ldots, t_k).$$

In particular, a term can be the following:

- c_i models a communication channel between honest parties;
- n_i, m_i, k_i are names and are used to model some data;
- x_i, y_i, z_i are variables that can represent any term, that is, any term can be bound to variables. Similarly as in case of the applied π-calculus [9];
- e_i is a cache entry;
- Finally, f is a function with arity k and is used to construct terms and to model cryptographic primitives, and messages. Complex messages are modelled by the function *tuple* with k terms: $tuple(t_1,\ldots, t_k)$, which we abbreviate as $(t_1,\ldots, t_k)$. The function symbol with arity zero is a constant;
- int_i ranges over special functions for modelling non-negative integers. Formally, let 0 be the base element of set $\mathcal{I}$, and is formally defined as the function named by 0. Each further integer is defined as a constructor function named by 1, 2, etc. Let the function $inc(int_i)$ be the function that increases the integer int_i by one. Numbers 1, 2, ... are modelled by functions $inc(0)$, $inc(1)$, ..., respectively. The relation between these intergers is defined by $int_i < inc(int_i)$ and $int_i = int_i$;

The internal operation of communication entities in the system is modelled by *processes*. Processes can be specified with the following syntax, and inductive definition:

$P, Q, R ::=$	*Processes*
$\overline{c}\langle t\rangle.P$	send
$c(x).P$	receive
$P \mid Q$	parallel composition
$P[\ ]Q$	enabled-action choice
$\nu n.P$	restriction
$I(y_1,\ldots,y_n)$	recursive definition
$[t_i = t_j]P$ else Q	if-else equal
$[int_i \geq int_j]P$ else Q	if-else larger or equal
$[int_i > int_j]P$ else Q	if-else larger
$[t_i = t_j]P$	if equal
$[int_i \geq int_j]P$	if larger or equal
$[int_i > int_j]P$	if larger
nil	does nothing
$let\ (x = t)\ in\ P$, $let\ (e = t)\ in\ P$	let

- The process $\overline{c}\langle t\rangle.P$ represents the sending of message t on channel c, followed by the execution of P. Process $c(x).P$ represents the receiving of some message, which is bound to x in P.
- In the composition $P \mid Q$, processes P and Q run in parallel. Each may interact with the other on channels known to both, or with the outside world, independently of the other. For example, the communication between the sending process $\overline{c}\langle t\rangle.P$ and receiving process $c(x).P$ can be described as the parallel composition $\overline{c}\langle t\rangle.P \mid c(x).Q$.
- A choice $P\ [\ ]\ Q$ can behave either as P or Q depending on the first visible/invisible action of P and Q. If the first action of P is enabled but the first action of Q's is not then P is chosen, and vice versa. In case both actions are enabled the behavior is the same as a non-deterministic choice.
- A restriction $\nu n.P$ is a process that makes a new, private (restricted) name n, and then behaves as P. The scope of n is restricted to P, and is available only for the process within its scope. A private channel c restricted to P is defined by $\nu c.P$, which does not allow for the attackers to eavesdrop on the channel.

- A typical way of specifying infinite behavior is by using parametric recursive definitions, like in the π-calculus [17]. Here $I(y_1, \ldots, y_n)$ is an identifier (or invocation) of arity n. We assume that every such identifier has a unique, possibly recursive, definition $I(x_1, \ldots, x_n) \stackrel{def}{=} P$ where the x_i's are pairwise distinct. The intuition is that $I(y_1, \ldots, y_n)$ behaves as P with each x_i replaced by y_i, respectively.
- In processes $[t_i = t_j]P$ *else* Q; $[int_i \geq int_j]P$ *else* Q; and $[int_i > int_j]P$ *else* Q: if $(t_i = t_j)$, $(int_i \geq int_j)$, and $(int_i > int_j)$, respectively, then process P is "activated", else they behave as Q. When Q is the **nil** process, we simply remove the else branch from the processes.
- The process **nil** does nothing, and is used to model the termination of a process behavior.
- Finally, *let* $(x = t)$ *in* P (or *let* $(e = t)$ *in* P) means that every occurrence of x (or e) in P is bound to t.

We adopt the notion of *environment*, well-known in process algebra, which is used to model the attacker(s) who can obtain the (publicly) exchanged messages, and can modify them. Moreover, we adopt the notation of the extended process and active substitution in the applied π-calculus [9] to model the information the attacker(s) (or the environment) is getting to know during the system run. The definition of the *extended process* is as follows:

A, B, C	::=	extended process
P		plain process
$A \mid B$		parallel composition
$\nu n.A$		name restriction
$\nu x.A$		variable restriction
$\{t/x\}$		active substitution

- P is a plain network we already discussed above.
- $A|B$ is a parallel composition of two extended process.
- $\nu n.A$ is a restriction of the name n to A.
- $\nu x.A$ is a restriction of the variable x to A.
- $\{t/x\}$ means that the binding of t to x, denoted by $\{t/x\}$, is applied to any process that is in parallel composition with $\{t/x\}$. Intuitively, the binding applies to *any* process that comes into contact with it. To restrict the binding $\{t/x\}$ to a process P, we use the variable restriction νx over $(\{t/x\} \mid P)$, namely, $\nu x.\,(\{t/x\} \mid P)$. Using this, the equivalent definition of process $\overline{c}\langle t\rangle.P$ can be given by $\nu x.(\overline{c}\langle x\rangle.P \mid \{t/x\})$. Active substitutions are always assumed to be cycle-free, that is, the set of bindings will never get into an infinite loop, to make the calculus well-defined.

  We write $fv(A)$, $bv(A)$, $fn(A)$, and $bn(A)$ for the sets of free and bound variables and free and bound names of A, respectively. These sets are defined as follow:

$$fv(\{t/x\}) \stackrel{def}{=} fv(t) \cup \{x\},\ fn(\{t/x\}) \stackrel{def}{=} fn(t)$$

$$bv(\{t/x\}) \stackrel{def}{=} \emptyset,\ bn(\{t/x\}) \stackrel{def}{=} bn(t)$$

  The concept of bound and free values is similar to local and global scope in programming languages. The scope of names and variables are delimited by binders $c(x)$ (i.e., input) and νn or νx (i.e., restriction). The set of bound names $bn(A)$ contains every name n which is under restriction νn inside A. The set of bound variables $bv(A)$ consists of all those variables x occurring in A that are bound by restriction νx or input $c(x)$. Further, we define the set of free names and the set of free variables. The set of free names in A, denoted by $fn(A)$,

consists of those names n occurring in A that are not a restricted name. The set of free variables $fv(A)$ contains the variables x occurring in A which are not a restricted variable (νx) or input variable ($c(x)$). A plain process P is closed if it contains no free variable. An extended process is closed when every variable x is either bound or defined by an active substitution.

As in the applied π-calculus, a frame (φ) is an extended process built up from the **nil** process and active substitutions of the form $\{t/x\}$ by parallel composition and restrictions. Formally, the frame $\varphi(A)$ of the extended process , $A = \nu n_1 \ldots n_k(\{t_1/x_1\} \mid \ldots \mid \{t_n/x_n\} \mid P)$, is $\nu n_1 \ldots n_k(\{t_1/x_1\} \mid \ldots \mid \{t_n/x_n\})$. The domain of the frame $\varphi(A)$ (denoted by $dom(A)$) is the set $\{x_1, \ldots, x_n\}$.

Intuitively, the frame $\varphi(A)$ accounts for the static knowledge exposed by A to its environment, but not for dynamic behavior. The frame allows access to terms which the environment cannot construct. For instance, after the term t (not available for the environment) are output in P resulting in $P' \mid \{t/x\}$, t becomes available for the enviroment. Finally, let σ range over substitutions (i.e., variable bindings). We write σt for the result of applying σ to the variables in t.

2.1.1 Labeled transition system ($\xrightarrow{\alpha}$)

The operational semantics for processes is defined as a labeled transition system $(\mathcal{P}, \mathcal{G}, \longrightarrow)$ where $\mathcal{P}$ represents a set of extended processes, $\mathcal{G}$ is a set of labels, and $\longrightarrow \subseteq \mathcal{P} \times \mathcal{G} \times \mathcal{P}$.

Specifically, the labeled semantics defines a ternary relation, written $A \xrightarrow{\alpha} B$, where α is a label of the form τ, $c(t)$, $\overline{c}\langle x\rangle$, $\nu x.\overline{c}\langle x\rangle$ where x is a variable of base type and t is a term. The transition $A \xrightarrow{\tau} B$ represents a silent move that are used to model the internal operation/computation of processes. These internal operations, such as the verification steps made on the received data, are not visible for the outside world, hence, to the attacker(s). The transition $A \xrightarrow{c(t)} B$ means that the process A performs an input of the term t from the environment on the channel c, and the resulting process is B. The label $\overline{c}\langle x\rangle$ is for output action of a free variable x. Finally, the label α is $\nu x.\overline{c}\langle x\rangle$ when a term is output on c. In the following, we give some examples for labeled transitions:

(Silent transition rules for processes:)

(Let1)	$let\ x = t\ in\ P \xrightarrow{\tau} P\{t/x\}$
(Let2)	$let\ e = t\ in\ P \xrightarrow{\tau} P\{t/e\}$
(IfElse1)	$[t_i = t_j]P\ else\ Q \xrightarrow{\tau} P$ (if $t_i = t_j$)
(IfElse2)	$[t_i = t_j]P\ else\ Q \xrightarrow{\tau}$ **nil** (if $t_i \neq t_j$)
(IfElse3)	$[int_i > int_j]P\ else\ Q \xrightarrow{\tau} P$ (if $int_i > int_j$)
(IfElse4)	$[int_i > int_j]P\ else\ Q \xrightarrow{\tau}$ **nil** (if $int_i \leq int_j$)
(IfElse5)	$[int_i \geq int_j]P\ else\ Q \xrightarrow{\tau} P$ (if $int_i \geq int_j$)
(IfElse6)	$[int_i \geq int_j]P\ else\ Q \xrightarrow{\tau}$ **nil** (if $int_i < int_j$)
(If1)	$[t_i = t_j]P \xrightarrow{\tau} P$ (if $t_i = t_j$)
(If2)	$[t_i = t_j]P \xrightarrow{\tau}$ **nil** (if $t_i \neq t_j$)
(If3)	$[int_i > int_j]P \xrightarrow{\tau} P$ (if $int_i > int_j$)
(If4)	$[int_i > int_j]P \xrightarrow{\tau}$ **nil** (if $int_i \leq int_j$)
(If5)	$[int_i \geq int_j]P \xrightarrow{\tau} P$ (if $int_i \geq int_j$)
(If6)	$[int_i \geq int_j]P \xrightarrow{\tau}$ **nil** (if $int_i < int_j$)
(Com)	$\overline{c}\langle t\rangle.P \mid c(x).Q \xrightarrow{\tau} P \mid Q\{t/x\}$

Let1-2 bind a variable to a term in a process. Rules *IfElse1-6* check the relation of two terms or integers, and say that if the condition holds. Rules *If1-6* is the corresponding rules for *IfElse1-6* when Q is the **nil** process. Besides these internal computations, the reduction relation usually is

used to model communication between a sender and a receiver process. This is specified by the rule (Com).

Next, we review the rules for output/input actions borrowed from the applied π-calculus. Rule (In) says that when a term t is received, it is bound to every occurence of x in P. Rule (Out) is related to the output of a not restricted name. Rule (Open) defines the output of a restricted name.

(Action transition rules for processes:)

(In) $\quad c(x).P \xrightarrow{c(t)} P\{t/x\}$

(Out) $\quad \overline{c}\langle u\rangle.P \xrightarrow{\overline{c}\langle u\rangle} P$

(Open) $\quad \dfrac{A \xrightarrow{\overline{c}\langle u\rangle} A', u \neq c}{\nu u.A \xrightarrow{\nu u.\overline{c}\langle u\rangle} A'}$

(Scope) $\quad \dfrac{A \xrightarrow{\alpha} A',\ u \notin \alpha}{\nu u.A \xrightarrow{\alpha} \nu u.A'}$

(Par) $\quad \dfrac{A \xrightarrow{\alpha} A',\ bv(\alpha) \cap fv(B) = bn(\alpha) \cap fn(B) = \emptyset}{A|B \xrightarrow{\alpha} A'|B}$

(Struct) $\quad \dfrac{A \equiv B, B \xrightarrow{\alpha} B', B' \equiv A'}{A \xrightarrow{\alpha} A'}$

From rule (Open) and the fact that $\nu x.(\overline{c}\langle x\rangle.P \mid \{t/x\})$ is equivalent to $\overline{c}\langle t\rangle.P$, we have two addional output rules

$$(\textbf{\textit{Out-1}}) \quad \overline{c}\langle t\rangle.P \xrightarrow{\nu x.\overline{c}\langle x\rangle} \{t/x\} \mid P \qquad (\textbf{\textit{Out-2}}) \quad \nu n.(\overline{c}\langle t\rangle.P) \xrightarrow{\nu x.\overline{c}\langle x\rangle} \nu n.(\{t/x\} \mid P);$$

For instance, based on the labeled transition system we have the following transitions:

$$\overline{c}\langle t_1\rangle.\overline{c}\langle t_2\rangle.P \xrightarrow{\nu x_1.\overline{c}\langle x_1\rangle} \{t_1/x_1\} \mid \overline{c}\langle t_2\rangle.P \xrightarrow{\nu x_2.\overline{c}\langle x_2\rangle} \{t_1/x_1\} \mid \{t_2/x_2\} \mid P$$

After sending the terms t_1 and t_2 on public channel c (modeled by action transitions $\xrightarrow{\nu x_1.\overline{c}\langle x_1\rangle}$ and $\xrightarrow{\nu x_2.\overline{c}\langle x_2\rangle}$ respectively), t_1 and t_2 become available for the environment (attacker), which fact is specified by the active substitutions $\{t_1/x_1\}$ and $\{t_2/x_2\}$.

After sending the terms t_1 and t_2 on public channel c (modeled by action transition $\xrightarrow{\nu x_1.\overline{c}\langle x_1\rangle}$ and $\xrightarrow{\nu x_2.\overline{c}\langle x_2\rangle}$ respectively), t_1 and t_2 become available for the environment (attacker), which fact is specified by the active substitutions $\{t_1/x_1\}$ and $\{t_2/x_2\}$.

Similarly as in [9], the set of function symbols $\sum$ is equipped with an *equational theory Eq*, that is, a set of equations of the form $t_1 = t_2$, where terms t_1, t_2 are defined over $\sum$. This allows us to define cryptographic primitives and operations, such as one-way hash function, MAC computation, encryption, decryption, and digital signature generation/verification, etc. For instance,
tuple: The constructor function **Tuple** models a tuple of n terms t_1, t_2,..., t_n. We write the function as

$$\mathit{Tuple}(t_1, t_2, \ldots, t_n)$$

We abbreviate it simply as $(t_1, t_2, \ldots, t_n)$ in the rest of the paper.

We introduce the destructor functions $\boldsymbol{i}$ that returns the i-th element of a tuple of n elements, where $i \in \{1, \ldots, n\}$:

$$i(t_1, t_2, \ldots, t_n) = t_i$$

We model the keyed hash or MAC function with symmetric key K with the binary function **MAC**. The MAC verification is defined in the form of an equation.

$$MAC(t, K);\ CheckMAC(MAC(t,K),\ K) = ok$$

Function MAC computes the message authentication code of message t using secret key K. The shared key between node l_i and l_j is modelled by function $K(l_i, l_j)$. The MAC verification defined by the function $CheckMAC$ is successful (returns a special name ***ok***) if the keys match each other.

We model the one-way hash function with the function **Hash** with one attribute. The

$$Hash(t)$$

function that computes the hash value of message t. Note that the hash function do not have any inverse counterpart because they are one-way functions.

2.2 Labeled bisimilarity

In this subsection, we give the definition of labeled bisimilarity, also known in [9], that says if two extended processes are equivalent, meaning that their behavior cannot be distinguished by an observer which can eavesdrop on communications.

Let the extended process A be $\{^{t_1}/_{x_1}\} \mid \ldots \mid \{^{t_n}/_{x_n}\} \mid P_1 \mid \ldots \mid P_n$. The *frame* φ of A is the parallel composition $\{^{t_1}/_{x_1}\} \mid \ldots \mid \{^{t_n}/_{x_n}\}$ that models all the information which is output so far by the process A. In particular, this information is the terms $t_1, \ldots, t_n$.

Definition 1. ***(Static equivalence for extended processes)*** *Two extended processes A_1 and A_2 are statically equivalent, denoted as $A_1 \approx_s A_2$, if their frames are statically equivalent. Two frames φ_1 and φ_2 are statically equivalent if they includes the same number of active substitutions and same domain; and any two terms that are equal in φ_1 are equal in φ_2 as well. Intuitively, this means that the outputs of the two processes cannot be distinguished by the environment.*

Definition 2. *Labeled bisimilarity ($\approx_l$) is the largest symmetric relation $\mathcal{R}$ on closed extended networks, such that $A_1 \mathcal{R} A_2$ implies*

- *$A_1 \approx_s A_2$;*
- *if $A_1 \xrightarrow{\tau} A_1'$, then $A_2 \xrightarrow{\tau}{}^{*} A_2'$ and $A_1' \mathcal{R} A_2'$ for some A_2';*
- *if $A_1 \xrightarrow{\alpha} A_1'$ and $fv(\alpha) \subseteq dom(A_1) \wedge bn(\alpha) \cap fn(A_2) = \emptyset$, then $\exists\ A_2'$ such that $A_2 \xrightarrow{\tau}{}^{*} \xrightarrow{\alpha} \xrightarrow{\tau}{}^{*} A_2'$ and $A_1'\ \mathcal{R}\ A_2'$, where $dom(A_1)$ denotes the domain of A_1.*

Intuitively, this means that the outputs of the two extended processes cannot be distinguished by the environment. In particular, the first point means that at first A_1 and A_2 are statically equivalent; the second point says that A_1 and A_2 remains statically equivalent after internal reduction steps. Finally, the third point says that if process A_1 outputs/inputs something then process A_2 ables to output/input the same thing, and the "target states" A_1' and A_2' they reach after that remain statically equivalent. Here, $\xrightarrow{\tau}{}^{*}$ models the sequential execution of some internal reduction steps.

3 $crypt_{time}$: Extending *crypt* with timed syntax and semantics

In this subsection, we propose a time extension to *crypt*, denoted by $crypt_{time}$. Our calculus is tailored for the the verification of security protocols, especially for verifying protocols that need to cache data, such as the transport protocols for wireless sensor networks. The design methodology of $crypt_{time}$ is based on the terminology proposed in the previous works in timed calculus [14, 6], and based on the syntax and semantics of the well-known timed automata. The main difference between our and those methods is that we focusing on extending *crypt*, which is differ (more

complex) from the calculus used in those related works, and we propose a new definition called *timed labeled bisimilarity* for proving the existence of the timing attacks against security protocols.

The concept of $crypt_{time}$ is based on the basic concept of timed automata, hence, the correctness of $crypt_{time}$ comes from the correctness of the timed automata because the semantic of $crypt_{time}$ is equivalent to the semantic of the timed automata, and we show that each process in $crypt_{time}$ has an associated timed automaton.

3.1 Basic time concepts

First of all, we provide some notations and notions related to clocks and time construct, borrowed from the well-known concept of timed automata. Assume a set $\mathcal{C}$ of nonnegative real valued variable called clocks. A valuation over $\mathcal{C}$ is a mapping $v : \mathcal{C} \mapsto \mathbb{R}^{\geq 0}$ assigning nonnegative real values to clocks. For a time value $d \in \mathbb{R}^{\geq 0}$ let $v + d$ denote the valuation such that $(v + d)(x_c) = v(x_c) + d$, for each clock $x_c \in \mathcal{C}$.

The set $\Phi(C)$ of clock constraints is generated by the following grammar:

$$\phi ::= true \mid false \mid x_c \sim N \mid \phi_1 \wedge \phi_2 \mid \neg\phi$$

where ϕ ranges over $\Phi(C)$, $x_c \in \mathcal{C}$, N is a natural, $\sim \in \{<, \leq, \geq, >\}$. We write $v \vDash \phi$ when the valuation v satisfies the constraint ϕ. Formally, $v \vDash true$; $v \vDash x_c \sim N$ iff $v(x_c) \sim N$; $v \vDash \phi_1 \wedge \phi_2$ iff $v \vDash \phi_1 \wedge v \vDash \phi_2$.

In the following we turn to define the following timed-process for $crypt_{time}$

$$\begin{array}{l} A_t ::= A \quad | \quad \alpha^* \prec A_t \quad | \quad \phi \hookrightarrow A_t \quad | \, \phi \triangleright A_t \quad | \quad \|C_R\| A_t \quad | \quad A_t^1 \, [\,] \, A_t^2 \quad | \\ \qquad (A_t^1 | A_t^2) \quad | \quad X_t \end{array}$$

We will discuss the meaning of $crypt_{time}$ processes by showing the connection between the modelling elements of timed automata and $crypt_{time}$. For this purpose, we recall the definition of timed automaton: a timed automaton Aut is defined by the tuple $(\mathcal{L}, l_0, \sum, \mathcal{C}, \mathcal{I}nv, \kappa, E)$, where

- $\mathcal{L}$ is a finite set of locations and l_0 is the initial location;
- $\sum$ is a set of actions that range over act;
- $\mathcal{C}$ is a finite set of clocks;
- $\mathcal{I}nv$:$\mathcal{L} \mapsto \Phi(\mathcal{C})$ is a function that assigns location to a formula, called a location invariant, that must hold at a given location;
- κ: $\mathcal{L} \mapsto 2^{\mathcal{C}}$ is the set of clock resets to be performed at the given locations;
- $E \subseteq \mathcal{L} \times \sum \times \Phi(\mathcal{C}) \times \mathcal{L}$ is the set of edges. We write $l \xrightarrow{act,\phi} l'$ when $(l, act, \phi, l') \in \longrightarrow$, where act, ϕ are the action and the time constraint defined on the edge.

Let us denote the set of processes in $crypt_{time}$ by $\mathbf{A}_{time}$, and we let A_t range over processes in $\mathbf{A}_{time}$. In $crypt_{time}$, each timed-process A_t^l corresponds to a location l in timed automaton, such that there is an initial process $A_t^{l_0}$ for location l_0. The set of actions $\sum$ corresponds to the set of actions known in *crypt*. The set of clocks to be reset at a given location l, $\kappa(l)$, is defined by the construct $\|C_R\| A_t^l$, where C_R is the set of clocks to be reset at the beginning of A_t^l. The location invariant at the location l corresponds to the construct $\phi \triangleright A_t^l$, and the edge guard can be defined by $\phi \hookrightarrow A_t^l$. More specifically,

- A_t can be an extended process A without any time construct.
- $\alpha^* \prec A_t$ represents the process A_t of which α^* is the first (not timed) action. Note that α^* can be $\nu x.\overline{c}\langle x\rangle$, $\overline{c}\langle u\rangle$, $c(t)$, and the silent action τ. For instance, if A_t is $c(t).P$, where P is the plain process in *crypt*, then α^* is $c(t)$.

- $\phi \hookrightarrow A_t$ represents the time guard, and says that the first action α^* of A_t is performed in case the guard (time constraint) ϕ holds. This process intends to model the edge $l \xrightarrow{\alpha,\phi} l'$ in the timed automaton syntax, where A_t corresponds to l, while the explicit appearance of the target location l' is omitted in the process.

- $\phi \triangleright A_t$ represents time invariant over A_t. Like in timed automaton, this means that the system cannot "stay" in process A_t once time constraint ϕ becomes invalid. If it cannot move from this process via any transition, then it is a deadlock situation. Invariant can be used to model timeout.

- in the timed process $\|C_R\|A_t$, the clocks in the set C_R are reset within the process A_t. We move the clock resetting from edge to the target state like in [6].

- A_t^1 [ ] A_t^2, and $A_t^1 \mid A_t^2$ describe the non-deterministic choice, first-action choice, and the parallel composition of two processes, respectively.

- X_t is a process variable to which one of the processes $\phi \hookrightarrow A_t$, $\phi \triangleright A_t$, $\|C_R\|A_t$ can be bound. Note that differ from [17], for our problem we restrict process variables to be only those processes that have time constructs defined on it. The reason we do this is that we want to avoid the recursive process invocation for extended processes, which may lead to infinite invocation cycle (e.g., $A = \{t/x\} \mid A$, where the process variable is abound by A), hence it is not well-defined. We allow recursive invocation for only plain processes (P) because (i) they describe the behavior of the system which should include recursive behavior, and (ii) the process variables (X_t) in them are guarded by an action (input, output, comparison) which prevents from an infinite invocation. Finally, for our problem, restricting X_t to one of the processes $\phi \hookrightarrow A_t$, $\phi \triangleright A_t$, $\|C_R\|A_t$ is sufficient.

The formal semantics of $crypt_{time}$ also follows the semantics of the timed automata. Namely, a state s is defined by the pair (A_t, v), where v is the clock valuation at the location of label A_t with the time issues defined at the location. The initial state s_0 consists of the initial process and initial clock valuation, (A_t^0, v_0). Note that the initial process A_t^0 is the initial status of a system behavior, while v_0 typically contains the clocks in the reset state. The operational semantics of $crypt_{time}$ is defined by a timed transition system (TTS).

A timed transition system can be seen as the labeled transition system extended with time constructs. In our model we adopt the concept of [6].

Definition 3. *Let $\sum$ be the set of actions. A time transition system is defined as the tuple $TTS = (\mathcal{S}, \sum \times \mathbb{R}^{\geq}, s_0, \longrightarrow_{TTS}, \mathcal{U})$ where*

- *$\mathcal{S}$ is a set of states, and s_0 is an initial state.*

- *$\longrightarrow_{TTS} \subseteq \mathcal{S} \times (\sum \times \mathbb{R}^{\geq 0}) \times \mathcal{S}$ is the set of timed labeled transition. A transition is defined between the source and target state, and the label of the transition is composed of the actions and the time stamp (duration) of the action. When $(\alpha^*, d) \in \sum \times \mathbb{R}^{\geq 0}$ we denote the transition from s to s' by $s \xrightarrow{\alpha^*,d}_{TTS} s'$.*

- *$\mathcal{U} \subseteq \mathbb{R}^{\geq 0} \times \mathcal{S}$ is the **until** predicate, and is defined at a state s with a time duration d. Whenever $(d, s) \in \mathcal{U}$ we use the notation $\mathcal{U}_d$.*

The timed transition system TTS should satisfy the two axioms Until and Delay (in both cases $\Longrightarrow$ denotes logical implication):

Until $\quad \forall\, d, d' \in \mathbb{R}^{\geq 0}, \mathcal{U}_d(s) \wedge (d' < d) \Longrightarrow \mathcal{U}_{d'}(s)$

Delay $\quad \forall\, d \in \mathbb{R}^{\geq 0}, s \xrightarrow{\alpha^*,d}_{TTS} s'$ for some $s' \Longrightarrow \mathcal{U}_d(s)$

These two axioms define formally the meaning of the notion delay and until. Basically, axiom ***Until*** says that if the system stays in state s until d time units then it also stays in this state before d. While the axiom ***Delay*** says that if the system performs an action α at time d then it must wait until d. Note that the meaning of until differs from time invariant, because in case of until, the system waits (stay idled) at least d time units in a state (location, if talking about automata), whilst invariant says that the system must leave the state (location) upon d time units have elapsed (if it cannot move from the state then we get deadlock).

We define the satisfaction predicate $\models$, $\models \subseteq \Phi(\mathcal{C})$, on clock constraints. For each $\phi \in \Phi(\mathcal{C})$ we use the shorthand $\models v(\phi)$ iff v satisfies ϕ, for all valuation v. The set of past closed constraint, $\overline{\Phi(\mathcal{C})} \subseteq \Phi(\mathcal{C})$, is used for defining semantics of location invariant, $\forall\, v \in \mathcal{V}$, $d \in \mathbb{R}^{\geq 0}$: $\models (v+d)(\phi) \implies \models (v)(\phi)$. Intuitively, this says that if the valuation $v+d$, which is defined as $v(x_c)+d$ for all clocks x_c, satisfies the constraint ϕ then so does v. We adopt the variant of time automata used in [6], where location invariant and clock resets are defined as functions ∂ and κ assigning a set of clock constraints $\overline{\Phi(\mathcal{C})}$ and a set of clocks to be reset $\mathcal{R}(\mathcal{C})$, respectively, to a $crypt_{time}$ process.

We adopt the variant of time automata used in [6], where location invariant and clock resets are defined as functions ∂ and κ assigning a set of clocks constraint $\overline{\Phi(\mathcal{C})}$ and a set of clocks to be reset $\mathcal{R}(\mathcal{C})$ to a $crypt_{time}$ process, respectively.

The interpretation (semantics) of $crypt_{time}$ is composed of the rule describing action moves and the rule defining the time passage at a state.

$$\text{(T-pass)}\ \frac{\models (v[rst:\kappa(A_t)]+d)(\partial(A_t))}{\mathcal{U}_d(A_t,v)}\ ; \quad \text{(T-Act)}\ \frac{(\phi \hookrightarrow A_t) \xrightarrow{\alpha^*} A'_t,\ \models (v[rst:\kappa(A_t)]+d)(\partial(A_t)\wedge\phi)}{(\phi \hookrightarrow A_t, v) \xrightarrow{\alpha^*,d}_{TTS} (A'_t, v[rst:\kappa(A_t)]+d)}$$

The rule (T-pass) describes the time passage at the same location. It says that if the system stays at process A_t until d time units, then the valuation $v+d$, after resetting the clocks in $\kappa(A_t)$, satisfies the invariant $\partial(A_t)$. Rule (T-Act) is concerning with the timed action move of a system from process $\phi \hookrightarrow A_t$ to process A'_t via action α^*. It says that there is a timed transition from state $(\phi \hookrightarrow A_t, v)$ to state (A'_t, v') with $v' = v[rst:\kappa(A_t)] + d$, if there is an edge $(\phi \hookrightarrow A_t) \xrightarrow{\alpha^*} A'_t$, such that v' satisfies the invariant at process A_t and the guard on the edge. Note v' is the valuation v in which clocks in $\kappa(A_t)$ are set to 0, and increased by d time units.

Definition 4. *We extend the definition of free and bound variable to the set of clock variables in processes A_t. The set of free variable and bound variable of A_t, $fvc(A_t)$ and $bvc(A_t)$, respectively, is the least set satisfying*

- $fvc(A) = \emptyset$*: The pure extended process contains no clock variables.*
- $fvc(\alpha^* \prec A_t) = fvc(A_t)$*: The set of free clock variables is not affected by action.*
- $fvc(\phi \hookrightarrow A_t) = clock(\phi) \cup fvc(A_t)$*: Edge guards contains free clock variables.*
- $fvc(\phi \triangleright A_t) = clock(\phi) \cup fvc(A_t)$*: Invariant contains free clock variables.*
- $fvc(\|C_R\|A_t) = fvc(A_t) \backslash C_R$*: Clocks to be reset are bound clock variables.*
- $fvc(A_t^1\ [\ ]\ A_t^2) = fvc(A_t^1) \cup fvc(A_t^2)$*: Union of free clock variables.*
- $fvc(A_t^1 \mid A_t^2) = fvc(A_t^1) \cup fvc(A_t^2)$*: Union of free clock variables.*

and for bound clock variables we have

- $bvc(A) = \emptyset$*: The pure extended process contains no clock variables.*
- $bvc(\alpha^* \prec A_t) = bvc(A_t)$*: The set of bound clock variables is not affected by action.*

- $bvc(\phi \hookrightarrow A_t) = bvc(A_t)$*: Edge guards contains no bound clock variables.*
- $bvc(\phi \triangleright A_t) = bvc(A_t)$*: Invariant contains no bound clock variables.*
- $bvc(\|C_R\|A_t) = bvc(A_t) \cup C_R$*: Clocks to be reset are bound clock variables.*
- $bvc(A_t^1[\ ]\ A_t^2) = bvc(A_t^1) \cup bvc(A_t^2)$*: Union of bound clock variables.*
- $bvc(A_t^1 \mid A_t^2) = bvc(A_t^1) \cup bvc(A_t^2)$*: Union of bound clock variables.*

Recall that the recursive process invocation X_t is a process variable, defined as $X_t \stackrel{def}{=} P(x_1, x_2, \ldots, x_n)$, for a plain process P, and describes a recursive process invocation. Since in recursive process invocations X_t only binds plain processes, it does not contain any free/bound clock variables. The reason that the set of clock variables is divided to bound and free parts is to avoid conflict of clock valuations. For instance, let us consider the process $x_c \leq 8 \triangleright (\|x_c\|\ A_t)$, in which the clock x_c is reset wich affects the invariant $x_c \leq 8$. Further, in the parallel composition $(\|x_c\|\ A_t) \mid (x_c \leq 8 \triangleright A'_t)$ the clock variable x_c is the shared variable of the two processes, however, the reset of x_c affects the behavior of process $(x_c \leq 8) \triangleright A'_t$, which is undesirable since the operation semantics of a process also depends on the behavior of the environment (that is hard to control).

Hence, we define the notion of process with non-conflict of clock variables, using the following inductive definition and the predicate *ncv*:

1. $ncv(A)$; **2.** $ncv(X_t)$; **3.** $ncv(\alpha^* \prec A_t)$ iff $ncv(A_t)$; **4.** $ncv(\|C_R\|\ A_t)$ iff $ncv(A_t)$;

5. $ncv(\phi \hookrightarrow A_t)$; **6.** $ncv(\phi \triangleright A_t)$: in both cases, iff $ncv(A_t) \wedge (clock(\phi) \cap \kappa(A_t) = \emptyset)$

7. $ncv(A_t^1\ [\ ]\ A_t^2)$ iff $ncv(A_t^1) \wedge ncv(A_t^2) \wedge (\kappa(A_t^1) \cap fvc(A_t^2) = \emptyset) \wedge (\kappa(A_t^2) \cap fvc(A_t^1) = \emptyset)$

8. $ncv(A_t^1 \mid A_t^2)$ iff $ncv(A_t^1) \wedge ncv(A_t^2) \wedge (\kappa(A_t^1) \cap fvc(A_t^2) = \emptyset) \wedge (\kappa(A_t^2) \cap fvc(A_t^1) = \emptyset)$

Rule 1 holds because an extended process A does not include any clock variable. Rule 2 says that the recursive process invocation of plain processes is non-conflict because a plain process does not contain clock variables. Rule 3 comes from the fact that action α^* is free from clock variables. Rule 4 says that if clock resettings are placed outside (outermost) all invariant and guard constructs then it does not cause conflict. Rules 5 and 6 says that if guard and invariant construct are placed outside then their clock variables cannot be reset within A_t, to avoid conflict. Finally, rules 7-8 are concerning with the cases of choice and parallel composition.

In the following, for each $crypt_{time}$ process we add rules that associate each process to the invariant and resetting function ∂ and κ, respectively. For the function κ we have:

k1. $\kappa(A) = \emptyset$; **k2.** $\kappa(\alpha^* \prec A_t) = \emptyset$; **k3.** $\kappa(\|C_R\|\ A_t) = C_R \cup \kappa(A_t)$;

k4. $\kappa(\phi \hookrightarrow A_t) = \kappa(A_t)$; **k5.** $\kappa(\phi \triangleright A_t) = \kappa(A_t)$; **k6.** $\kappa(A_t^1\ [\ ]\ A_t^2) = \kappa(A_t^1) \cup \kappa(A_t^2)$;

k7. $\kappa(A_t^1 \mid A_t^2) = \kappa(A_t^1) \cup \kappa(A_t^2)$.

Rule *k1* is true because an extended process does not contain any clock; rule *k2* the set of clocks to be reset for $\alpha^* \prec A_t$ is empty because there is no clock reset construct defined on it; Rule *k3* says that the set of clocks to be reset in $\kappa(\|C_R\|\ A_t)$ is C_R and the clock resets occur in A_t; The clock resets of choices and parallel composition constructs are the union of the clock resets. For the invariant function ∂ we have:

i1. $\partial(A) = true$; **i2.** $\partial\ (\alpha^* \prec A_t) = true$; **i3.** $\partial(\|C_R\|\ A_t) = \partial(A_t)$;

i4. $\partial\ (\phi \hookrightarrow A_t) = \partial\ (A_t)$; **i5.** $\partial\ (\phi \triangleright A_t) = \partial\ (A_t) \wedge \phi$; **i6.** $\partial\ (A_t^1\ [\ ]\ A_t^2) = \partial(A_t^1) \vee \partial(A_t^2)$;

i7. $\partial\ (A_t^1 \mid A_t^2) = \partial(A_t^1) \vee \partial(A_t^2)$.

Rule *i1* says that the invariant predicate of a extended process is true because it does not include clocks; Rules *i2*, *i3* and *i4* say that there is no any invariant construct defined on these processes; Rule *i5* says that the invariant of process $\phi \triangleright A_t$ is the intersection of ϕ and the invariant predicate in A_t. The invariant predicate of choices and parallel composition is the disjunction of the predicates (in rules *i6*-*i7*).

In addition, we give the rules for processes that are correspond to automata edges:

t1. $\alpha^* \prec A_t \overset{\alpha^*,\ true}{\longrightarrow} A_t$; **t2.** $\phi \hookrightarrow (\alpha^* \prec A_t) \overset{\alpha^*,\ \phi}{\longrightarrow} A_t$; **i3.** $\phi \hookrightarrow (\phi' \hookrightarrow (\alpha^* \prec A_t)) \overset{\alpha^*, \phi\wedge\phi'}{\longrightarrow} A_t$

t4. $(\phi \hookrightarrow (\alpha^* \prec (\phi' \triangleright A_t^1))\ )\ [\ ]\ A_t^2 \overset{\alpha^*,\ \phi\wedge\phi'}{\longrightarrow} A_t^1$; **t5.** $(\phi \hookrightarrow (\alpha^* \prec (\phi' \triangleright A_t^1))\ )\ |\ A_t^2 \overset{\alpha^*,\ \phi\wedge\phi'}{\longrightarrow} A_t^1$;

t6. $\|C_R\|(\phi \hookrightarrow (\alpha^* \prec A_t\ )) \overset{\alpha^*,\ \phi}{\longrightarrow} A_t$; **t7.** $\phi' \triangleright (\phi \hookrightarrow (\alpha^* \prec A_t\ )) \overset{\alpha^*,\ \phi\wedge\phi'}{\longrightarrow} A_t$.

It is very important to note that the edge $\overset{\alpha^*,\ \phi}{\longrightarrow}$ does not change the validity of the *ncv* property to be invalid. The following theorem says that the notion of associated timed automata to each A_t is well-defined

Theorem 1. *For each process A_t such that ncv(A_t), the associated timed automata, denoted by $\mathcal{T}(A_t)$, is indeed a timed automata.*

Now we turn to discuss the *operational semantics* of $crypt_{time}$, in terms of the semantics of timed automata. The TTS of a $crypt_{time}$ process A_t with the initial clock valuation v_0, denoted by $TTS(A_t, v_0)$, is defined by the tuple $(A_t \times v, \sum \times \mathbb{R}^{\geq 0}, (A_t, v_0), \longrightarrow_{TTS}, \mathcal{U})$ where $\longrightarrow_{TTS}$ and $\mathcal{U}$ are the least set satisfying the following rules

u1. $\mathcal{U}_d(A, v)$; **u2.** $\mathcal{U}_d(\alpha^* \prec A_t, v)$; **u3.** $\mathcal{U}_d(\phi \hookrightarrow A_t, v)$ if $\mathcal{U}_d(A_t, v)$;

u4. $\mathcal{U}_d(\|C_R\|\ A_t, v)$ if $\mathcal{U}_d(A_t, v[rst : C_R])$; **u5.** $\mathcal{U}_d(\phi \triangleright A_t, v)$ if $\mathcal{U}_d(A_t, v) \wedge \models (v + d)(\phi)$;

u6. $\mathcal{U}_d(A_t^1\ [\ ]\ A_t^2, v)$ if $\mathcal{U}_d(A_t^1, v)$; **u7.** $\mathcal{U}_d(A_t^1 \mid A_t^2, v)$ if $\mathcal{U}_d(A_t^1, v)$;

u8. $\mathcal{U}_d(X, v)$ if $\mathcal{U}_d(P[P/X_t], v)$;

Rules (*u1*-*u2*) are the Until axioms for the states (A, v) and $(\alpha^* \prec A_t, v)$. In *u3* the system stays in the state $(\phi \hookrightarrow A_t, v)$ until d time units, if this is valid to the state (A_t, v) as well. Rules (*u4*-*u5*) come from the definition of the clock reset and invariant. In rule (*u4*) $v[rst : C_R]$ represents the clock valuation v where the clocks in C_R are reset. Rules (*u6*-*u8*) say that the system stays until d time units at the state with $A_t^1\ [\ ]\ A_t^2$, $A_t^1 \mid A_t^2$, and $A_t^1 \oplus_p A_t^2$, if it stays d time in the state with one of the two processes A_t^1 and A_t^2. Rule *u9* is concerned with the until predicate for (recursive) process variable X_t, which comes directly from the definition of recursive process invocation. Note that P is a plain process defined in *crypt*. The timed transition (action) rules for $crypt_{time}$ are given as follows:

a1. $(\alpha^* \prec A_t, v) \overset{\alpha^*,d}{\longrightarrow}_{TTS} (A_t, v + d)$;

a2. $(\|C_R\|\ A_t,\ v) \xrightarrow{\alpha^*,d}_{TTS} (A'_t,\ v')$ if $(A_t,\ v[rst : C_R]) \xrightarrow{\alpha^*,d}_{TTS} (A'_t,\ v')$;

a3. $(\phi \hookrightarrow A_t,\ v) \xrightarrow{\alpha^*,d}_{TTS} (A'_t,\ v')$ if $(A_t,\ v) \xrightarrow{\alpha^*,d}_{TTS} (A'_t,\ v') \wedge (v+d)(\phi)$;

a4. $(\phi \triangleright A_t,\ v) \xrightarrow{\alpha^*,d}_{TTS} (A'_t,\ v')$ if $(A_t,\ v) \xrightarrow{\alpha^*,d}_{TTS} (A'_t,\ v') \wedge (v+d)(\phi)$;

a5. $(A^1_t\ [\ ]\ A^2_t,\ v) \xrightarrow{\alpha^*,d}_{TTS} (A'^1_t,\ v')$ if $(A^1_t,\ v) \xrightarrow{\alpha^*,d}_{TTS} (A'^1_t,\ v')$;

a6. $(A^1_t \mid A^2_t,\ v) \xrightarrow{\alpha^*,d}_{TTS} (A'^1_t \mid norst(A^2_t),\ v')$ if $(A^1_t,\ v) \xrightarrow{\alpha^*,d}_{TTS} (A'^1_t,\ v')$;

a7. $(X_t,\ v) \xrightarrow{\alpha^*,d}_{TTS} (P',\ v')$ if $(P[P/X_t],\ v) \xrightarrow{\alpha^*,d}_{TTS} (P',\ v')$.

Rule *a1* says that after performing action α^* with d time units the system gets to the process A_t with the clock valuation after d time units elapsed. $v[rst : C_R]$ in the rule *a2* represents the valuation where the clocks in C_R are reset. In the rules *a3* and *a4* the timed transition can be performed if $(v+d)(\phi)$ holds, which means that the valuation $v+d$ must satisfy the clock guard ϕ. Rules *a5-a6* describe the case when process A^1_t is activated (the rules for activating A^2_t are similar). In *a6* to avoid conflict of clock variable, we assume that after performing the transition, process A^2_t cannot start with clock reset. The last rule is the action rule for the recursive process variable X_t. It can be proven, based on the rules *u1-u8* and *a1-a7*, that $TTS(A_t,\ v_0)$ satisfies axioms *Until* and *Delay*, hence, it is well defined.

Theorem 2. *For all $crypt_{time}$ process A_t and for all closed valuation v_0, $TTS(A_t,\ v_0)$ is indeed the times transition system defined in timed automata.*

3.2 Renaming of clock variables

We show that the process with *ncv* property is preserved by clock renaming, hence, the restriction to process without conflict of clock variables is harmless [6]. Let predicate rn represents clock renaming, we have

n1. $rn(A) = A$; **n2.** $rn(\ \alpha^* \prec A_t) = rn(A_t)$; **n3.** $rn(\phi \hookrightarrow A_t) = rn(\phi) \hookrightarrow rn(A_t)$;

n4. $rn(\|C_R\|\ A_t) = \|\mathcal{F}(C_R)\|\ rn[\mathcal{F}](A_t)$; **n5.** $rn(\phi \triangleright A_t) = rn(\phi) \triangleright rn(A_t)$;

n6. $rn(A^1_t\ [\ ]\ A^2_t) = rn(A^1_t) \cup rn(A^2_t)$; **n7.** $rn(A^1_t \mid A^2_t) = rn(A^1_t) \cup rn(A^2_t)$;

where $\mathcal{F}$: $C_R \mapsto V$ is bijective function mapping a set of clock C_R to an another set of clocks $V \in \mathcal{C}$ (i.e., renaming), such that the resulted clock set V does not contain the renamed clocks in invariant and guard within A_t, formally, $V \cap rn(fcv(A_t) \backslash C_R)$. Note that the traditional renaming of names and variables defines renaming of bound variables and names in processes, we allow renaming of free clock variables in rules n3 and n5, since the clocks in invariant and guard are free by definition.

Now based on the rules of renaming we add new rules for structural equivalent resulted from renaming, denoted by $\equiv_{rn}$. Two process A^1_t and A^2_t are structurally equivalent by renaming of clock variables, $A^1_t \equiv_{rn} A^2_t$ if they are (stuctural equivalent to) the left and right side of the rules n1-n7, respectively.

s1. if $A^1_t \equiv_{rn} A^2_t$ then (i) $\alpha^* \prec A^1_t \equiv_{rn} \alpha^* \prec A^2_t$; (ii) $\phi \hookrightarrow A^1_t \equiv_{rn} \phi \hookrightarrow A^2_t$
(iii) $\phi \triangleright A^1_t \equiv_{rn} \phi \triangleright A^2_t$;

s2. if $A_t^1 \equiv_{rn} A_t^2$ and $A_t'^1 \equiv_{rn} A_t'^2$ then (i) $A_t^1 \ [\,]\ A_t'^1 \equiv_{rn} A_t^2 \ [\,]\ A_t'^2$;
(ii) $A_t^1 \mid A_t'^1 \equiv_{rn} A_t^2 \mid A_t'^2$;

s3. $\|C_R^1\|A_t^1 \equiv_{rn} \|C_R^2\|A_t^2$ if clocks in C_R^1 are renamed to C_R^2 ($\mathcal{F}$: $C_R^1 \mapsto C_R^2$) that the *ncv* property is preserved: $C_R^2 \cap fcv(\|C_R^1\|\ A_t^1) = \emptyset$ and $A_t^1 \equiv_{rn} A_t^2$;

In addition to the rules for renaming we can define the next structural equivalent rules for A_t

s4. $A_t^1 \ [\,]\ A_t^2 \equiv A_t^2 \ [\,]\ A_t^1$

s5. $(A_t^1 \ [\,]\ A_t^2) \ [\,]\ A_t^3 \equiv A_t^3 \ [\,]\ (A_t^1 \ [\,]\ A_t^2)$

s6. $(\phi^1 \hookrightarrow A_t^1) \ [\,]\ (\phi^2 \hookrightarrow A_t^2) \equiv (\phi^1 \vee \phi^2) \hookrightarrow (A_t^1 \ [\,]\ A_t^2)$

s7. $(\phi^1 \triangleright A_t^1) \ [\,]\ (\phi^2 \triangleright A_t^2) \equiv (\phi^1 \vee \phi^2) \triangleright (A_t^1 \ [\,]\ A_t^2)$

s8. $\mathbf{false} \hookrightarrow (\alpha^* \prec A_t) \equiv \mathbf{nil}$

s9. $\phi \hookrightarrow \mathbf{nil} \equiv \mathbf{nil}$

s10. $\phi^1 \triangleright(\ \phi^2 \triangleright A_t) \equiv (\phi^1 \wedge \phi^2) \triangleright A_t$

s11. $\phi^1 \hookrightarrow(\ \phi^2 \triangleright A_t) \equiv \phi^2 \triangleright(\ \phi^1 \hookrightarrow A_t)$

s12. $\phi \hookrightarrow(\ \|C_R\|\ A_t) \equiv \|C_R\|(\ \phi \hookrightarrow A_t)$, if $clock(\phi) \cap C_R = \emptyset$

s13. $\phi \hookrightarrow (A_t^1 \ [\,]\ A_t^2) \equiv (\phi \hookrightarrow A_t^1) \ [\,]\ (\phi \hookrightarrow A_t^2)$

s14. $\mathbf{true} \hookrightarrow A_t \equiv A_t$

s15. $\mathbf{true} \triangleright A_t \equiv A_t$

s16. $\phi^1 \triangleright(\ \phi^2 \triangleright A_t) \equiv (\phi^1 \wedge \phi^2) \triangleright A_t$

s17. $\phi \triangleright(\ \|C_R\|\ A_t) \equiv \|C_R\|(\ \phi \triangleright A_t)$, if $clock(\phi) \cap C_R = \emptyset$

s18. $\phi \triangleright (A_t^1 \ [\,]\ A_t^2) \equiv (\phi \triangleright A_t^1) \ [\,]\ (\phi \triangleright A_t^2)$

s19. $\|C_R\|\ A_t \equiv A_t$, if $fcv(A_t) \cap C_R = \emptyset$

s20. $\|C_R\|\ \|C_R'\|\ A_t \equiv \|C_R \cup C_R'\|\ A_t$

s21. $\|C_R\|\ (A_t^1 \ [\,]\ A_t^2) \equiv \|C_R\|\ A_t^1 \ [\,]\ \|C_R\|\ A_t^2$.

In the following we consider the parallel composition of two $crypt_{time}$ processes, $A_t^1 \mid A_t^2$. We discuss the bound (*bcv*) and free clock variables (*fvc*), the non-conflict of variable predicate (*ncv*), along with the axioms. First, we specifies the bound and free variables:

Definition 5. *We extend the definition of free and bound variables of A_t, such that the set of free and bound variables of A_t is the least set satisfying the following rules (with the previous given definitions)*

- $fvc(A_t^1 \mid A_t^2) = fvc(A_t^1) \cup fvc(A_t^2)$*: The free clock variables is the union of the parallel processes.*

- $fvc(norst(A_t)) = \kappa(A_t) \cup fvc(A_t)$: *The free clock variables of a the process* $norst(A_t)$ *is the union of the clock resets at* A_t *and its free clock variables.*
- $bvc(A_t^1 \mid A_t^2) = bvc(A_t^1) \cup bvc(A_t^2)$: *The bound clock variables is the union of the parallel processes.*
- $bvc(norst(A_t)) = bvc(A_t)$: *The free clock variables of a the process* $norst(A_t)$ *is the same as* A_t.

and the rules for non-conflict of variable (ncv) predicate:

Definition 6. *We extend the definition of predicatee ncv as follows*

- $ncv(norst(A_t))$ *if* $ncv(A_t)$ *holds.*
- $ncv(A_t^1 \mid A_t^2)$ *if* $ncv(A_t^1) \wedge ncv(A_t^2)$, $bvc(A_t^1) \cap var(A_t^2) = \emptyset \wedge bvc(A_t^2) \cap var(A_t^1) = \emptyset$.

The time constructs in case of parallel composition are defined as follows:

k8. $\kappa(A_t^1 \mid A_t^2) = \kappa(A_t^1) \cup \kappa(A_t^2)$; **k9.** $\kappa(norst(A_t)) = \emptyset$; **k10.** $\kappa(norst(\|C_R\|\ A_t)) = \emptyset \cup \kappa(A_t)$;

i8. $\partial(A_t^1 \mid A_t^2) = \partial(A_t^1) \wedge \partial(A_t^2)$; **i9.** $\partial(A_t) = \partial(norst(A_t))$.

The action transition for parallel composition are

a9. $(A_t^1 \mid A_t^2) \xrightarrow{\alpha^*,\phi} (A_t'^1 \mid norst(A_t^2))$ if $A_t^1 \xrightarrow{\alpha^*,\phi} A_t'^1$; **a10.** $norst(A_t^1) \xrightarrow{\alpha^*,\phi} A_t'^1$ if $A_t^1 \xrightarrow{\alpha^*,\phi} A_t'^1$.

Finally, we give the structural equivalence for the parallel composition, and name and variable restrictions.

s22. $A_t^1 \mid A_t^2 \equiv A_t^2 \mid A_t^1$
s23. $\phi \hookrightarrow A_t^1 \mid A_t^2 \equiv \phi \hookrightarrow (A_t^1 \mid A_t^2)$
s24. $(\|C\|\ A_t^1) \mid A_t^2 \equiv \|C\|\ (A_t^1 \mid A_t^2)$ if $C \cap fvc(A_t^2) = \emptyset$
s25. $(\partial \triangleright A_t^1) \mid A_t^2 \equiv \partial \triangleright (A_t^1 \mid A_t^2)$
s26. $(\nu k.\phi \triangleright A_t) \equiv \phi \triangleright (\nu k.A_t)$
s27. $(\nu x.\phi \triangleright A_t) \equiv \phi \triangleright (\nu x.A_t)$
s28. $(\nu k.\|C\|\ A_t) \equiv (\|C\|\nu k.A_t)$
s29. $(\nu x.\|C\|\ A_t) \equiv (\|C\|\nu x.A_t)$
s30. $(\nu k.\phi \hookrightarrow A_t) \equiv (\phi \hookrightarrow \nu k.A_t)$
s31. $(\nu x.\phi \hookrightarrow A_t) \equiv (\phi \hookrightarrow \nu x.A_t)$

Any process defined in $crypt_{time}$ can be expressed in a corresponding timed automata. To show this, first we adopt the notion *image-finite* and *finitely sorted* (borrowed from transition system theory). A timed automaton is image-finite if the set of outgoing edges of each state with the same action *act*. Formally, for each l and *act* the size of the set $\{l \xrightarrow{act,\ \phi} l' \mid l \in \mathcal{L}\}$ is finite. A timed automaton is finitely-sorted if the set of outgoing edges with the same action *act* of every state, $\{act \mid \exists\ l' \in \mathcal{L}\colon\ l \xrightarrow{act,\ \phi} l'\}$, is finite.

The associated timed automaton for a (initial) process A_t^0 can be constructed by associating the process A_t^0 to the initial location l_0, then each transition made by $A_t^0 \xrightarrow{\alpha^*,\ \phi} A_t^1$ can be defined in terms of timed automaton, $T = (\mathcal{S}, \sum, \mathcal{C}, l_0, \longrightarrow, \kappa, \partial)$, as follows:

$$A_t^0 = \|\kappa(l_0)\|\ \partial(l_0) \triangleright (\phi \hookrightarrow (\alpha^* \prec A_t^1))$$

In this process definition, A_t^0 corresponds to location l_0 of the timed automata at which the set of clocks to be reset is $\kappa(l_0)$, and on which the invariant $\partial(l_0)$ is defined. The edge from l_0 to l_1, $l_0 \xrightarrow{\alpha^*,\ \phi} l_1$, corresponds to the time construct $\phi \hookrightarrow (\alpha^* \prec A_t^1)$. Generally, for every subsequent process A_t^i after some transition steps from A_t^0 we have

$$A_t^i = \|\kappa(l_i)\|\ \partial(l_i) \triangleright (\phi \hookrightarrow (\alpha^* \prec A_t^{i+1}))$$

which corresponds to the edge $l_i \overset{\alpha^*,\ \phi}{\longrightarrow} l_{i+1}$ in T. For the more complex target process such as $A_t^{(i+1)_1}\ [\ ] \ldots [\ ]\ A_t^{(i+1)_n}$ we have

$$A_t^i = \|\kappa(l_i)\|\ \partial(l_i) \triangleright [\ ]_{j=1}^n(\phi_j \hookrightarrow (\alpha_j^* \prec A_t^{(i+1)_j}))$$

where A_t^i corresponds to location l_i (with the appropriate resets and invariant) and the subprocess $[\ ]_{j=1}^n(\phi_j \hookrightarrow (\alpha_j^* \prec A_t^{(i+1)_j}))$ corresponds to the edge from l_i to the location $l_{(i+1)_j}$ with label $(\alpha_j^*,\ \phi_j)$, $1 \leq j \leq n$, such that α_j^* is the first enabled action (due to the valid condition at l_i) among the n processes. In case there are more than one enabled action at the same time, it can be treated in the same way as the non-deterministic choices.

In case there is not any outgoing edge from l_i we have the following process definitions for each type of target process:

$$A_t^i = \|\kappa(l_i)\|\ \partial(l_i) \triangleright \mathbf{nil}$$

We omit the case where the target process is the parallel composition of other processes. Finally, we provide the notion of timed labeled bisimilarity that can be seen as a combination of a timed bisimilarity defined for timed automata [6], and the labeled bisimilarity defined in applied π-calculus.

The novel Definition 7 describes the *timed labeled bisimulation* for $crypt_{time}$ processes to prove timing attacks against security protocols.

Definition 7. (***Timed labeled bisimulation for crypt$_{time}$ processes***)
Let $TTS_i(A_t^i, v_0) = (\mathcal{S}_i,\ \sum \times \mathbb{R}^{\geq 0},\ s_0^i,\ \longrightarrow_{TTSi},\ \mathcal{U}^i)$, $i \in \{1, 2\}$ be two timed transition systems for $crypt_{time}$ processes. Timed labeled bisimilarity ($\approx_T$) is the largest symmetric relation $\mathcal{R}$, $\mathcal{R} \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ with $s_0^1\ \mathcal{R}\ s_0^2$, where each s_i is the pair of a closed $crypt_{time}$ process and a same initial valuation $v_0 \in \mathcal{V}^c$, $(A_t^i,\ v_0)$, such that $s_1\ \mathcal{R}\ s_2$ implies:

1. *$A^1 \approx_s A^2$;*
2. *if $s_1 \overset{(\tau,d)}{\longrightarrow}_{TTS1} s_1'$, then $\exists\ s_2'$ such that $s_2 \overset{(\tau,\sum d_i)}{\Longrightarrow}_{TTS2} s_2'$ and $s_1'\ \mathcal{R}\ s_2'$, and $d = f(\sum d_i)$ for some function f;*
3. *if $s_1 \overset{(\alpha,d)}{\longrightarrow}_{TTS1} s_1'$ and $fv(\alpha) \subseteq dom(A^1) \wedge bn(\alpha) \cap fn(A^2) = \emptyset$, then $\exists\ s_2'$ such that $s_2 \overset{(\alpha,\sum d_j)}{\Longrightarrow}_{TTS2} s_2'$ and $s_1'\ \mathcal{R}\ s_2'$, and $d = f(\sum d_j)$ for some function f. Again, $dom(A^1)$ represents the domain of A^1.*

Where the extended processes A^1 and A^2 are the "untimed" version of the processes A_t^1 and A_t^2, respectively, by removing all time constructs in them.

The arrow $\overset{\alpha}{\Longrightarrow}_{TTS}$ is the same as $\overset{\tau}{\longrightarrow}^*_{TTS}\ \overset{\alpha}{\longrightarrow}_{TTS}\ \overset{\tau}{\longrightarrow}^*_{TTS}$, where $\overset{\tau}{\longrightarrow}^*_{TTS}$ represents a series (formally, a transitive closure) of sequential transitions $\overset{\tau}{\longrightarrow}_{TTS}$. $\sum d_i$ on $\Longrightarrow_{TTS}$ is the sum of the time elapsed at each transition, and represents the total time elapsed during the sequence of transitions. Note that $fn(A_t^2)$ and $dom(A_t^1)$ is the same as $fn(A^2)$ and $dom(A^1)$. Moreover, a process A_t is closed if its untimed counterpart A is closed.

Intuitively, in case A_t^1 and A_t^2 represent two protocols (or two variants of a protocol), then this means that (i) the outputs of the two processes cannot be distinguished by the environment during their behaviors; (ii) the time that the protocols spend on the performed operations until they reach the corresponding points is in some relationship defined by a function f. Here f depends on the specific definition of the security property, for instance, it can return d itself, hence, the requirement for time consumption would be $d = \sum d_i$. In particular, the first point means that at first A_t^1 and A_t^2 are statically equivalent, that is, the environment cannot distinguish the behavior of the two protocols based on their outputs; the second point says that A_t^1 and A_t^2 remain statically equivalent after silent transition (internal reduction) steps. Finally, the third point says that the behavior of the two protocols matches in transition with the action α.

4 $crypt^{prob}_{time}$: The probabilistic timed calculus for cryptographic protocols

$crypt^{prob}_{time}$ is the extension of $crypt_{time}$ with probabilistic syntax and semantics. This is a new probabilistic timed calculus for cryptographic protocols, and to the best of our knowledge, it is the first of its kind. The definition of $crypt^{prob}_{time}$ is inspired by the syntax and semantics of the probabilistic extension of the applied π-calculus in [10], and the probabilistic automata in [6].

We extend the set of processes A_t defined in $crypt_{time}$ (Section 3) with the probabilistic choice. Let us denote the probabilistic timed process by A_{pt}, which is an extended process in *crypt* with time constructs and probabilistic choice: $A^1_{pt} \oplus_p A^2_{pt}$. Formally, we have the following probabilistic timed processes for $crypt^{prob}_{time}$:

$$A_{pt} ::= A \quad | \quad \alpha^* \prec_{pi} A_{pt} \quad | \quad \phi \hookrightarrow A_{pt} \quad | \quad \|C_R\|A_{pt} \quad | \quad A^1_{pt} \; [\,] \; A^2_{pt} \; | \quad A^1_{pt} \oplus_p A^2_{pt} \\ | \; (A^1_{pt} | A^2_{pt}) \quad | \quad X_{pt}$$

Process $A^1_{pt} \oplus_p A^2_{pt}$ behaves like A^1_{pt} with probability p, and with $(1-p)$ it behaves as A^2_{pt}. $\alpha^* \prec_\pi A_{pt}$ performs α^* as the first (not timed) action with the distribution π, at any time, and then it behaves like A_{pt}. We extend the definition of predicates *fvc* and *bvc* with free and bound clock variables in the probabilistic choice: $fvc(A^1_{pt} \oplus_p A^2_{pt})$, $bvc(A^1_{pt} \oplus_p A^2_{pt})$. The other rules are defined similarly as in case of timed processes but with replacing A_t by A_{pt} in them.

Definition 8. *We extend the definition of predicates fvc and bvc with free and bound clock variables in the probabilistic choice:* $fvc(A^1_{pt} \oplus_p A^2_{pt})$, $bvc(A^1_{pt} \oplus_p A^2_{pt})$. *The other rules are similar as in case of timed processes but* A_t *is replaces by* A_{pt}.

- $fvc(A) = \emptyset$: *The pure extended process contains no clock variables.*
- $fvc(\alpha^* \prec A_{pt}) = fvc(A_{pt})$: *The set of free clock variables is not affected by action.*
- $fvc(\phi \hookrightarrow A_{pt}) = clock(\phi) \cup fvc(A_{pt})$: *Edge guards contains free clock variables.*
- $fvc(\phi \triangleright A_{pt}) = clock(\phi) \cup fvc(A_{pt})$: *Invariant contains free clock variables.*
- $fvc(\|C_R\|A_t) = fvc(A_t) \backslash C_R$: *Clocks to be reset are bound clock variables.*
- $fvc(A^1_{pt} \; [\,] \; A^2_{pt}) = fvc(A^1_{pt}) \cup fvc(A^2_{pt})$: *Union of free clock variables.*
- $fvc(A^1_{pt} \oplus_p A^2_{pt}) = fvc(A^1_{pt}) \cup fvc(A^2_{pt})$: *Union of free clock variables.*
- $fvc(A^1_{pt} \mid A^2_{pt}) = fvc(A^1_{pt}) \cup fvc(A^2_{pt})$: *Union of free clock variables.*

and for bound clock variables

- $bvc(A) = \emptyset$: *The pure extended process contains no clock variables.*
- $bvc(\alpha^* \prec A_{pt}) = bvc(A_{pt})$: *The set of bound clock variables is not affected by action.*
- $bvc(\phi \hookrightarrow A_{pt}) = bvc(A_{pt})$: *Edge guards contains no bound clock variables.*
- $bvc(\phi \triangleright A_{pt}) = bvc(A_{pt})$: *Invariant contains no bound clock variables.*
- $bvc(\|C_R\|A_{pt}) = bvc(A_{pt}) \cup C_R$: *Clocks to be reset are bound clock variables.*
- $bvc(A^1_{pt} \; [\,] \; A^2_{pt}) = bvc(A^1_{pt}) \cup bvc(A^2_{pt})$: *Union of free clock variables.*
- $bvc(A^1_{pt} \oplus_p A^2_{pt}) = bvc(A^1_{pt}) \cup bvc(A^2_{pt})$: *Union of free clock variables.*

- $bvc(A^1_{pt} \mid A^2_{pt}) = bvc(A^1_{pt}) \cup bvc(A^2_{pt})$*: Union of free clock variables.*

X_{pt} is a process variable, defined as $X_{pt} \stackrel{def}{=} P(x_1, x_2, \ldots, x_n)$, for a plain process P, and describing recursive process invocation. Since X_{pt} only binds plain processes it does not have any free/bound clock variables.

For $crypt^{prob}_{time}$ the following rule is added to the definition of the predicate ncv (non-conflict of variables):

$$ncv(A^1_{pt} \oplus_p A^2_{pt}) \text{ iff } ncv(A^1_{pt}) \wedge ncv(A^2_{pt}) \wedge (\kappa(A^1_{pt}) \cap fvc(A^2_{pt}) = \emptyset) \wedge (\kappa(A^2_{pt}) \cap fvc(A^1_{pt}) = \emptyset)$$

For the functions κ and ∂ we have the following two additional rules:

$$\textbf{rk. } \kappa(A^1_{pt} \oplus_p A^2_{pt}) = \kappa(A^1_{pt}) \cup \kappa(A^2_{pt}); \textbf{ ri. } \partial\ (A^1_{pt} \oplus_p A^2_{pt}) = \partial(A^1_{pt}) \vee \partial(A^2_{pt});$$

In addition, we give probabilistic timed transition rules for $crypt^{prob}_{time}$ processes that are corresponding to the edges in probabilistic timed automata:

t8. $A_{pt} \xrightarrow{\alpha^*}_{\pi} A'_{pt}$ if $A_{pt} \xrightarrow{\alpha^*} \pi$ and $\pi(A'_{pt}) > 0$;

t9. $\alpha^* \prec A_{pt} \xrightarrow{\alpha^*,\ tt,\ 1}_{\pi} A_{pt}$ if $\alpha^* \prec A_{pt} \xrightarrow{\alpha^*} \pi$ and $\pi(A_{pt}) = 1$;

t10. $\phi \hookrightarrow (\alpha^* \prec A_{pt}) \xrightarrow{\alpha^*,\ \phi,\ 1}_{\pi} A_{pt}$; if $\phi \hookrightarrow (\alpha^* \prec A_{pt}) \xrightarrow{\alpha^*} \pi$ and $\pi(A_{pt}) = 1$;

t11. $\phi \hookrightarrow (\phi' \hookrightarrow (\alpha^* \prec A_{pt})) \xrightarrow{\alpha^*,\phi\wedge\phi',\ 1}_{\pi} A_{pt}$; if $\phi \hookrightarrow (\phi' \hookrightarrow (\alpha^* \prec A_{pt})) \xrightarrow{\alpha^*} \pi$ and $\pi(A_{pt}) = 1$;

t12. $(\phi \hookrightarrow (\alpha^* \prec A^1_{pt}\ ))\ [\ ]\ A^2_{pt} \xrightarrow{\alpha^*,\ \phi,\ 1}_{\pi} A^1_{pt}$; if $\phi \hookrightarrow (\alpha^* \prec A^1_{pt}\ ) \xrightarrow{\alpha^*} \pi$ and $\pi(A^1_{pt}) = 1$;

t13. $(\phi \hookrightarrow (\alpha^* \prec A^1_{pt}\ ))\ |\ A^2_{pt} \xrightarrow{\alpha^*,\ \phi,\ 1}_{\pi} A^1_{pt}$; if $\phi \hookrightarrow (\alpha^* \prec A^1_{pt}\ ) \xrightarrow{\alpha^*} \pi$ and $\pi(A^1_{pt}) = 1$;

t14. $\|C_R\|(\phi \hookrightarrow (\alpha^* \prec A_{pt})) \xrightarrow{\alpha^*,\ \phi,\ 1}_{\pi} A_{pt}$; if $\|C_R\|(\phi \hookrightarrow (\alpha^* \prec A_{pt}\ )) \xrightarrow{\alpha^*} \pi$ and $\pi(A_{pt}) = 1$;

t15. $\phi' \triangleright (\phi \hookrightarrow (\alpha^* \prec A_{pt}\ )) \xrightarrow{\alpha^*,\ \phi,\ 1}_{\pi} A_{pt}$; if $\phi' \triangleright (\phi \hookrightarrow (\alpha^* \prec A_{pt}\ )) \xrightarrow{\alpha^*} \pi$ and $\pi(A_{pt}) = 1$;

t16/a. $A^1_{pt} \oplus_p A^2_{pt} \xrightarrow{\alpha^*,\ p}_{\pi_1} A^{1'}_{pt}$ if $A^1_{pt} \xrightarrow{\alpha^*}_{\pi_1} A^{1'}_{pt}$ and $\pi_1(A^{1'}_{pt}) = p$;

t16/b. $A^1_{pt} \oplus_p A^2_{pt} \xrightarrow{\alpha^*,\ 1-p}_{\pi_2} A^{2'}_{pt}$ if $A^2_{pt} \xrightarrow{\alpha^*}_{\pi_2} A^{2'}_{pt}$ and $\pi_2(A^{2'}_{pt}) = 1 - p$;

t17. $(\phi \hookrightarrow (\alpha^* \prec A^1_{pt}\ )) \oplus_p A^2_{pt} \xrightarrow{\alpha^*,\ \phi,\ p}_{\pi_1} A^1_{pt}$ if $\phi \hookrightarrow (\alpha^* \prec A^1_{pt}) \xrightarrow{\alpha^*}_{\pi_1} A^1_{pt}$ and $\pi_1(A^1_{pt}) = p$.

Rule t8 says that from process A_{pt} we can reach process A'_{pt} by performing the action α^*, with the distribution π. $A_{pt} \xrightarrow{\alpha^*} \pi$ and $\pi(A'_{pt}) > 0$ says that from A_{pt} we can perform α^* according to the distribution π such that the probability of reaching A'_{pt} is larger than 0. Rule t9 says that from process $\alpha^* \prec A_{pt}$ we can reach A_{pt} with probability 1, and without an edge guard limitation, (ie., ϕ is true, denoted by **tt**). This rule is equivalent to (or comes from) the assumptions $\alpha^* \prec A_{pt} \xrightarrow{\alpha^*} \pi$ and $\pi(A_{pt}) = 1$. Rules t10 and t11 are similar to t9 except that the edge guards are not **tt** but ϕ and $\phi \wedge \phi'$. Rule t12 is concerned with the semantics of the choice construct, saying that if the process on the left side of the choice operator is activated first then the action α^* is

performed with probability 1. In case the processes in both sides of [ ] are enabled at the same time, then we talk about non-deterministic choice. Rule t13 describes the case for the parallel composition construct similarly as in the choice case. Rule t14 says that after resetting the clocks in C_R process $\phi \hookrightarrow (\alpha^* \prec A_{pt})$ proceeds to A_{pt} with probability 1. Rule t15 considers the case of clock invariant, while rules 16/a, 16/b and 17 describe the action rules for probabilistic choice.

The operational semantics of $crypt^{prob}_{time}$ can be constructed by compounding the semantics of $crypt_{time}$ and the probabilistic extension of the applied π-calculus [10], making it also respect the operational semantics of the probabilistic timed automata [14]. Similarly as in probabilistic timed automata, we define a state s in $crypt^{prob}_{time}$ that is composed of a probabilistic timed process A_{pt}, and a clock valuation v, namely, $s = (A_{pt}, v)$. In [6] the time passage and the timed action transitions are modelled with the predicate *until* $\mathcal{U}_d(s)$, and the action transition with a label $s \xrightarrow{\alpha^*(d)}_{PTTS} s'$. The action transition says that the action $\alpha^* = \alpha \cup \tau$ is performed at time d for some $d \in \mathbb{R}^{\geq 0}$, which means that the system stays at s until time d, and then leaves it. We use an equivalent interpretation, namely, performing either a visible α or invisible (silent) τ action consumes d time units. However, not like [6] where the idling time at s is d and executing an action takes no time, we interpret d as the time for executing action α^*, and there is no idling time at s before performing an action.

We extend the label on transitions with the distribution π of the action α^*, $s \xrightarrow{\alpha^*(d),\ \pi}_{PTTS} s'$. For $s_{i-1} = (A^{i-1}_{pt}, v_{i-1})$ and $s_i = (A^i_{pt}, v_i)$, a transition $s_{i-1} \xrightarrow{\alpha^*_{i-1}(d),\ \pi_{i-1}}_{PTTS} s_i$ is ***enabled*** if there is a labeled transition $A^{i-1}_{pt} \xrightarrow{\alpha^*_{i-1},\ \phi_{i-1},\ p}_{\pi_{i-1}} A^i_{pt}$ such that $v_i \models \phi_{i-1}$, $v_i \models inv(A^{i-1}_{pt}) \wedge inv(A^i_{pt})$, where $v_i = v_{i-1}[\kappa(A^{i-1}_{pt}) := 0] + d$, $inv(A^{i-1}_{pt})$ and $\kappa(A^{i-1}_{pt})$ are the invariant predicate as well as the set of clocks to be reset defined at A^{i-1}_{pt}, respectively. We denote the set of transitions enabled at state s by $EN(s)$, and the set of transitions that lead from s to a certain s' through a given action α^* by $EN(s, \alpha^*, s')$. A state $s = (A_{pt}, v)$ is called *terminal* iff $EN(s') = \emptyset$ for all $s' = (A_{pt}, v + d')$, $d' \in \mathbb{R}^{\geq 0}$.

Let us define a finite set of probability distributions, $\Pi = \{\pi_1, \ldots, \pi_n\}$, where each π_i specifies a distribution of transitions. The scheduler F chooses non-deterministically *the distribution of action transition steps*. The probability of performing a transition step from a state $s = (A_{pt}, v)$ is based on the defined distribution π.

An (probabilistic timed) action execution of $s_0 = (A^0_{pt}, v_0)$, denoted by $Exec_{s_0}$, is a finite (or infinite) sequence of transition steps

$$e = s_0 \xrightarrow{\alpha^*_0(d_0),\ \pi_0}_{PTTS} s_{n_1} \xrightarrow{\alpha^*_1(d_1),\ \pi_1}_{PTTS} \cdots \xrightarrow{\alpha^*_{k-1}(d_{k-1}),\ \pi_{k-1}}_{PTTS} s_{n_k} \cdots$$

where each transition $s \xrightarrow{\alpha^*_i(d_i),\ \pi_i}_{PTTS} s'$ in $Exec_s$ is modelled by $s \xrightarrow{\sum d_j = d_i}_{PTTS^*} \xrightarrow{\alpha^*_i,\ \pi}_{PTTS} s'$, in which every (time step or action) transition is *enabled*. We denote $\xrightarrow{\sum d_j = d}_{PTTS^*}$ as a series of time step transitions, where $\sum d_j = d$ means that the sum of time amounts of the time step transitions is d. In case the execution is finite we denote with e^{n_k}, for a finite n_k, where

$$e^{n_k} = s_0 \xrightarrow{\alpha^*_0(d_0),\ \pi_0}_{PTTS} s_{n_1} \xrightarrow{\alpha^*_1(d_1),\ \pi_1}_{PTTS} \cdots \xrightarrow{\alpha^*_{k-1}(d_{k-1}),\ \pi_{k-1}}_{PTTS} s_{n_k},$$

and denote the last state of e^{n_k} by $last(e^{n_k})$ (e.g., $last(e^{n_k}) = s_{n_k}$).

A scheduler F defined in $crypt^{prob}_{time}$ resolves both the non-deterministic and the probabilistic choices. F is a partial function from execution fragment F: $Exec_s \mapsto \Pi \cup \mathbb{R}^{>0}$. Given a scheduler F and an execution fragment e^{n_k} we assume that F is defined for e^{n_k} if and only if there exists a reachable state s such that $last(e^{n_k}) \xrightarrow{\alpha^*,\ \pi}_{PTTS} s$, for $\alpha^* \in \alpha \cup \{\tau\}$, or $last(e^{n_k}) \xrightarrow{d'}_{PTTS} s$, for $d' \in \mathbb{R}^{>0}$.

The execution fragments $Exec^F_{s_0}$ from s_0 according to a scheduler F is defined as the set of executions

$$s_0 \xrightarrow{\alpha_0^*(d_0),\ \pi_0}_{PTTS} s_{n_1} \xrightarrow{\alpha_1^*(d_1),\ \pi_1}_{PTTS} \cdots \xrightarrow{\alpha_{k-1}^*(d_{k-1}),\ \pi_{k-1}}_{PTTS} s_{n_k} \cdots$$

such that **(i)** whenever $F(e^{i-1}) = d'$, for $d' \in \mathbb{R}^{>0}$, meaning that a time step has been chosen at state s_{i-1}, and **(ii)** whenever F returns a distribution $\pi_{i-1} \in \Pi$: $F(e^{i-1}) = \pi_{i-1}$, there is an enabled transition $s_{i-1} \xrightarrow{\alpha_{i-1}^*,\ \pi_{i-1}}_{PTTS} s_i$, where $\pi_{i-1}(e^{i-1})>0$.

The probability $P^F(e^{n_k})$ of the execution fragment

$$e^{n_k} = s_0 \xrightarrow{\alpha_0^*(d_0),\ \pi_0}_{PTTS} s_{n_1} \xrightarrow{\alpha_1^*(d_1),\ \pi_1}_{PTTS} \cdots \xrightarrow{\alpha_{k-1}^*(d_{k-1}),\ \pi_{k-1}}_{PTTS} s_{n_k},$$

based on the scheduler F is defined as follows:

- if $n_k = 0$ then $P^F(e^{n_k}) = 1$.
- if $n_k \geq 1$ then $P^F(e^{n_k}) = P^F(e^{n_k-1}) * p$.

where the probability of the *enabled* transition with action $\widetilde{\alpha}_{n_k-1}$ from the state s_{n_k-1} to s_{n_k} is defined as

$$p = \frac{\sum_{pttr \in EN(s_{n_k-1},\ \widetilde{\alpha}_{n_k-1},\ s_{n_k})} (F(e^{n_k-1}))(pttr)}{\sum_{pttr \in EN(s_{n_k-1})} (F(e^{n_k-1}))(pttr)} \textit{if } \widetilde{\alpha}_{n_k-1} \in \alpha \cup \{\tau\}$$

The first point of the formula says that for action transitions the probability p is the ratio of (i) the total probability of the enabled transitions with action α_{n_k-1} from s_{n_k-1} to s_{n_k}, based on the scheduler F , and (ii) the total probability from the enabled transitions from s_{n_k-1}, based on F. Note that for action transitions $F(e^{n_k-1})$ returns some distribution $\pi_{n_k-1} \in \Pi$. The second point of the formula says that the probability of time passage steps is 1, hence, whenever a time step is chosen (non-deterministically) it will be performed. We note that the probability of the action transition going from s_{n_k-1} to s_{n_k}, labeled with α_{n_k-1} is *re-normalized* according to the transitions enabled in s_{n_k-1}. The reason of re-normalizing the probability is that the number of enabled transitions in s_{n_k-1} varies according to the validity of edge guards and location invariants [6].

The operational semantics of $crypt_{time}^{prob}$ is given by a probabilistic timed transition system (PTTS). The PTTS of a process A_{pt} with the initial clock valuation v_0 and scheduler F, denoted by $PTTS(A_{pt},\ v_0,\ F)$, is defined by the tuple $(\mathcal{S}_{pt},\ \sum \times \mathbb{R}^{\geq 0} \times \Pi,\ (A_{pt},\ v_0),\ \longrightarrow_{PTTS},\ \mathcal{U},\ F)$ where $\longrightarrow_{PTTS}$ and $\mathcal{U}$ are the least set satisfying the rules ***u1*-*u8*** in Section 3 where A_{pt} and X_{pt} takes place instead of A_t and X_t, and we also add a new rule for probabilistic choice:

$$\textbf{u9. } \mathcal{U}_d(A_{pt}^1 \oplus_p A_{pt}^2,\ v) \quad \text{if} \quad \mathcal{U}_d(A_{pt}^1,\ v) \vee \mathcal{U}_d(A_{pt}^2,\ v);$$

The probabilistic timed transition rules for $crypt_{time}^{prob}$ is the following:

$$\textbf{a1. } (\alpha^* \prec_\pi A_{pt},\ v) \xrightarrow{\alpha^*(d),\ \pi}_{PTTS} (A_{pt},\ v+d) \quad \text{if} \quad \alpha^* \prec_\pi A_{pt} \xrightarrow{\alpha^*,\ true}_\pi A_{pt};$$

$$\textbf{a2. } (\|C_R\|\ A_{pt},\ v) \xrightarrow{\alpha^*(d),\ \pi}_{PTTS} (A'_{pt},\ v') \quad \text{if} \quad (A_{pt},\ v[rst : C_R]) \xrightarrow{\alpha^*(d),\ true}_\pi (A'_{pt},\ v');$$

$$\textbf{a3. } (\phi \hookrightarrow A_{pt},\ v) \xrightarrow{\alpha^*(d),\ \pi}_{PTTS} (A'_{pt},\ v') \quad \text{if} \quad (A_{pt},\ v) \xrightarrow{\alpha^*(d),\ \phi}_\pi (A'_{pt},\ v') \wedge (v+d)(\phi);$$

$$\textbf{a4. } (\phi \triangleright A_{pt},\ v) \xrightarrow{\alpha^*(d),\ \pi}_{PTTS} (A'_{pt},\ v') \quad \text{if} \quad (A_{pt},\ v) \xrightarrow{\alpha^*(d),\ true}_\pi (A'_{pt},\ v') \wedge (v+d)(\phi);$$

a5. $(A_{pt},\, v) \xrightarrow{\alpha^*(d),\ \pi}_{PTTS} (\phi \triangleright A'_{pt},\, v')$ if $(A_{pt},\, v) \xrightarrow{\alpha^*(d),\ true}_{\pi} (A'_{pt},\, v') \wedge (v+d)(\phi)$;

a6. $(A^1_{pt}\ [\ ]\ A^2_{pt},\, v) \xrightarrow{\alpha^*(d),\ \pi}_{PTTS} (A^{1'}_{pt},\, v')$ if $(A^1_{pt},\, v) \xrightarrow{\alpha^*(d),\ true}_{\pi} (A^{1'}_{pt},\, v')$;

a7/a. $(A^1_{pt} \oplus_p A^2_{pt},\, v) \xrightarrow{\alpha^*(d),\ \pi(p)}_{PTTS} (A^{1'}_{pt},\, v')$ if $A^1_{pt} \oplus_p A^2_{pt} \xrightarrow{\alpha^*,\ true}_{\pi(p)} A^{1'}_{pt}$

a7/b. $(A^1_{pt} \oplus_p A^2_{pt},\, v) \xrightarrow{\alpha^*(d),\ \pi(1-p)}_{PTTS} (A^{2'}_{pt},\, v')$ if $A^1_{pt} \oplus_p A^2_{pt} \xrightarrow{\alpha^*,\ true)}_{\pi(1-p)} A^{2'}_{pt}$

a8. $(A^1_{pt} \mid A^2_{pt},\, v) \xrightarrow{\alpha^*(d),\ \pi}_{PTTS} (A^{1'}_{pt} \mid norst(A^2_{pt}),\, v')$ if $(A^1_{pt},\, v) \xrightarrow{\alpha^*(d),\ true}_{\pi} (A^{1'}_{pt},\, v')$;

a9. $(X_{pt},\, v) \xrightarrow{\alpha^*(d),\ \pi}_{PTTS} (P',\, v')$ if $(P[P/X_{pt}],\, v) \xrightarrow{\alpha^*(d),\ true}_{\pi} (P',\, v')$.

In rule *a2* $v' = v[rst : C_R] + d$, and in the rest rules $v' = v + d$. $v[rst : C_R]$ represents the valuation v where the clocks in C_R are reset. Each rule should be interpreted that the PTTS transition on the left side can be performed if there is an edge in a corresponding automaton. For instance, rule *a1* applies if there is an edge $\alpha^* \prec_\pi A_{pt} \xrightarrow{\alpha^*,\ true}_{\pi} A_{pt}$ in the correspoding automaton. Rule *a1* says that after performing action α^* with d time units the system gets to the process A_{pt} with the clock valuation after d time units elapsed. Rule *a2* says that by the time $\|C_R\|$ A_{pt} proceeds to A_{pt}, the clocks in C_R will have been reset. In the rules *a3* and *a4* the timed transition can be performed if $(v+d)(\phi)$ holds, which means that the valuation $v + d$ must satisfy the clock guard ϕ. Rules *a5-a6* describe the case when process A^1_{pt} is activated (the rules for activating A^2_{pt} are similar). $\pi(p)$ and $\pi(1-p)$ in rules *a7/a-b* mean that in distribution π the first and second transitions (edges) are chosen with probability p and $(1-p)$. In *a8* to avoid conflict of clock variables, we require that after performing the transition process A^2_{pt} cannot perform resetting at the beginning. The last rule is the action rule for recursive process variable X_{pt}. It can be proven, based on the rules *u1-u9* and *a1-a9*, that probabilistic timed transition system of $crypt^{prob}_{time}$ satisfies axioms *Until* and *Delay*, hence, it is well defined.

Finally, we provide two novel bisimilarity definitions, called *weak prob-timed labeled bisimilarity* and *strong prob-timed labeled bisimilarity*, for $crypt^{prob}_{time}$ which enable us to prove or refute the security of probabilistic timed systems. In our proposed *bisimulations*, we extend the static equivalence with time and probabilistic elements. The meaning of *weak* is that in this paper we want to examine whether the attackers can distinguish the behavior of two processes, based on the information they can *observe*. Hence, in weak prob-timed labeled bisimulation, we do not require the equivalence of the probability of two action traces, because practically an observer cannot distinguish if an action is performed with 1/2 or 1/3 probability. The definition of *strong prob-timed labeled bisimulation* is stricter, since it also distinguishes two processes based on the probability of their corresponding action traces.

As mentioned before, the definition, notations and notion of static equivalence and frames are kept unchanged in $crypt^{prob}_{time}$. The definition of probabilistic labeled bisimilarity is based on the well-known static equivalent from the applied π-calculus [9].

Definition 9. (***Weak prob-timed labeled bisimulation for*** $crypt^{prob}_{time}$ ***states***)
Let $PTTS_i(A^i_{pt}, v_0, F) = (\mathcal{S}_i,\, \alpha \times \mathbb{R}^{\geq 0} \times \Pi,\, s^i_0,\, \longrightarrow_{PTTSi},\, \mathcal{U}^i,\, F)$, $i \in \{1, 2\}$ *be two probabilistic timed transition systems for* $crypt^{prob}_{time}$ *processes. Weak prob-timed labeled bisimilarity* ($\approx_{pt}$) *is the largest symmetric relation* $\mathcal{R}$, $\mathcal{R} \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ *with* $s^1_0\ \mathcal{R}\ s^2_0$, *where each* s^i *is the pair of a closed* $crypt^{prob}_{time}$ *process and a same initial valuation* $v_0 \in \mathcal{V}^c$, $(A^i_{pt},\, v_0)$, *such that* $s_1\ \mathcal{R}\ s_2$ *implies:*

1. $A^1 \approx_s A^2$;

2. *if* $s_1 \xrightarrow{\tau(d),\ \pi}_{PTTS1} s_1'$ *for a scheduler* F*, then* $\exists\ s_2'$ *such that* $s_2 \overset{\tau(\sum d_i),\ \pi_i}{\Longrightarrow}_{PTTS2} s_2'$ *for the same* F*, with* $d = f(\sum d_i)$ *for some function* f*, and* $s_1'\ \mathcal{R}\ s_2'$*;*

3. *if* $s_1 \xrightarrow{\alpha(d),\ \pi}_{PTTS1} s_1'$ *for a scheduler* F *and* $fv(\alpha) \subseteq dom(A^1) \wedge bn(\alpha) \cap fn(A^2) = \emptyset$*, then* $\exists\ s_2'$ *such that* $s_2 \overset{\alpha(\sum d_j),\ \pi_i}{\Longrightarrow}_{PTTS2} s_2'$ *for the same* F*, with* $d = f(\sum d_j)$ *for some function* f*, and* $s_1'\ \mathcal{R}\ s_2'$*. Again,* $dom(A^i)$ *represents the domain of* A^i*;*

and vice versa. A^1 *and* A^2 *are the extended processes we get by removing all the probabilistic and timed elements from* A^1_{pt} *and* A^2_{pt}*, respectively.*

Basically, the definition of *weak prob-timed labeled bisimulation* is the same as the definition of *timed labeled bisimulation*, but it is valid to probabilistic timed processes. The interpretation of *weak prob-timed labeled bisimulation* is similar to the case of *timed labeled bisimulation*.

Given a scheduler F, similarly as in [6], $\sigma Field^F$ is the smallest sigma field on $Exec^F$ that contains the basic cylinders $e\uparrow$, where $e \in Exec^F$. The probability measure $Prob^F$ is the unique measure on $\sigma Field^F$ such that $Prob^F(e\uparrow) = P^F(e)$.

Definition 10. ***(Strong prob-timed labeled bisimilarity for crypt$^{prob}_{time}$ states)*** *Let* $PTTS_i(A^i_{pt}, v_0, F) = (\mathcal{S}_i,\ \alpha \times \mathbb{R}^{\geq 0} \times \Pi,\ s^i_0,\ \longrightarrow_{PTTSi},\ \mathcal{U}^i,\ F)$*,* $i \in \{1, 2\}$ *be two probabilistic timed transition systems for crypt$^{prob}_{time}$ processes. Strong prob-timed labeled bisimilarity (*$\approx_{spt}$*) is the largest symmetric relation* $\mathcal{R}$*,* $\mathcal{R} \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ *with* $s^1_0\ \mathcal{R}\ s^2_0$*, where each* s^i *is the pair of a closed crypt$^{prob}_{time}$ process and a same initial valuation* $v_0 \in \mathcal{V}^c$*,* $(A^i_{pt},\ v_0)$*, such that* $s_1\ \mathcal{R}\ s_2$ *implies:*

1. $A^1 \approx_s A^2$*;*

2. *if* $s_1 \xrightarrow{\tau(d),\ \pi}_{PTTS1} s_1'$ *for a scheduler* F*, then* $\exists\ s_2'$ *such that* $s_2 \overset{\tau(\sum d_i),\ \pi_i}{\Longrightarrow}_{PTTS2} s_2'$ *and*

   (a) $Prob^F(s_1 \xrightarrow{\tau(d),\ \pi}_{PTTS1} s_1') = Prob^F(s_2 \overset{\tau(\sum d_i),\ \pi_i}{\Longrightarrow}_{PTTS2} s_2')$*;*

   (b) $d = f(\sum d_i)$ *for some function* f*;*

   (c) $s_1'\ \mathcal{R}\ s_2'$*.*

3. *if* $s_1 \xrightarrow{\alpha(d),\ \pi}_{PTTS1} s_1'$ *and* $fv(\alpha) \subseteq dom(A^1) \wedge bn(\alpha) \cap fn(A^2) = \emptyset$*, then* $\exists\ s_2'$ *such that* $s_2 \overset{\alpha(\sum d_j),\ \pi_i}{\Longrightarrow}_{PTTS2} s_2'$ *and*

   (a) $Prob^F(s_1 \xrightarrow{\alpha(d),\ \pi}_{PTTS1} s_1') = Prob^F(s_2 \overset{\alpha(\sum d_j),\ \pi_i}{\Longrightarrow}_{PTTS2} s_2')$*;*

   (b) $d = f(\sum d_i)$ *for some function* f*;*

   (c) $s_1'\ \mathcal{R}\ s_2'$*;*

and vice versa. The extended processes A^1 *and* A^2 *are the processes* A^1_{pt} *and* A^2_{pt} *after removing probabilistic and time constructs, respectively.*

$fn(A^2_{pt})$ and $dom(A^1_{pt})$ is the same as $fn(A^2)$ and $dom(A^1)$. Moreover, a process A_{pt} is closed if its untimed and probability free counterpart A is closed. Intuitively, in case A^1_{pt} and A^2_{pt} represent two protocols (or two variants of a protocol), then Definition 10 means that (i) the outputs of the two processes cannot be distinguished by the environment during their behaviors; (ii) the time that the protocols spend on the performed operations until they reach the corresponding points is in some relationship defined by a function f. Here f depends on the specific definition of the security property, for instance, it can return d itself, hence, the requirement for time consumption would be $d = \sum d_i$; (iii) the probability of the two corresponding executions $s_1 \xrightarrow{\tau(d),\ \pi}_{PTTS1} s_1'$ and $s_2 \overset{\tau(\sum d_i),\ \pi_i}{\Longrightarrow}_{PTTS2} s_2'$, and $s_1 \xrightarrow{\alpha(d),\ \pi}_{PTTS1} s_1'$ and $s_2 \overset{\alpha(\sum d_j),\ \pi_i}{\Longrightarrow}_{PTTS2} s_2'$ are equal.

In particular, the first point means that at first A^1_{pt} and A^2_{pt} are statically equivalent, that is, the environment cannot distinguish the behavior of the two protocols based on their outputs; the second point says that A^1_{pt} and A^2_{pt} remain statically equivalent after silent transition (internal reduction) steps. Finally, the third point says that the behavior of the two protocols matches in transition with the action α.

In order to reason about the difference between the two systems (or two variants of a system) where for instance, in one system a clock, say x_c, is reset at some point during the operation, while in the another system x_c is not reset, we need a bisimilarity definition that distinguishes the current valuation of the clocks. The following definitions enable us to reason about the difference of clock valuations in two systems:

Definition 11. (***Weak clock-val labeled bisimulation for crypt$^{prob}_{time}$ states***)
Let $PTTS_i(A^i_{pt}, v_0, F) = (\mathcal{S}_i,\ \alpha \times \mathbb{R}^{\geq 0} \times \Pi,\ s^i_0,\ \longrightarrow_{PTTSi}, \mathcal{U}^i,\ F)$, $i \in \{1, 2\}$ be two probabilistic timed transition systems for crypt$^{prob}_{time}$ processes. Weak clock-val labeled bisimilarity ($\approx_{cv}$) is the largest symmetric relation $\mathcal{R}$, $\mathcal{R} \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ with $s^1_0\ \mathcal{R}\ s^2_0$, where each s^i is the pair of a closed crypt$^{prob}_{time}$ process and a same initial valuation $v_0 \in \mathcal{V}^c$, $(A^i_{pt},\ v_0)$, such that $s_1\ \mathcal{R}\ s_2$ implies:

1. $A^1 \approx_s A^2$;
2. *if $s_1 \xrightarrow{\tau(d),\ \pi}_{PTTS1} s'_1$ for a scheduler F, then $\exists\ s'_2$ such that $s_2 \overset{\tau(\sum d_i),\ \pi_i}{\Longrightarrow}_{PTTS2} s'_2$ for the same F, with $v'_1 = f(v'_2)$, for some function f, and $s'_1\ \mathcal{R}\ s'_2$;*
3. *if $s_1 \xrightarrow{\alpha(d),\ \pi}_{PTTS1} s'_1$ for a scheduler F and $fv(\alpha) \subseteq dom(A^1) \wedge bn(\alpha) \cap fn(A^2) = \emptyset$, then $\exists\ s'_2$ such that $s_2 \overset{\alpha(\sum d_j),\ \pi_i}{\Longrightarrow}_{PTTS2} s'_2$ for the same F, with $v'_1 = f(v'_2)$, for some function f, and $s'_1\ \mathcal{R}\ s'_2$, $dom(A^i)$ represents the domain of A^i;*

and vice versa. A^1 and A^2 are the extended processes we get by removing all the probabilistic and timed elements from A^1_{pt} and A^2_{pt}, respectively. v'_1 and v'_2 are the clock valuations at the states s'_1 and s'_2, respectively. The function f can be applied to a subset of the entire set of clocks defined in the second system.

We also add the corresponding strong clock-val labeled bisimilarity definition for the strong prob-timed labeled bisimilarity by adding the requirement $v'_1 = f(v'_2)$ in the second and third points of the definition. Of course, if needed we can also provide the definition in which we keep both the requirements $d = f_1(\sum d_i)$ and $v'_1 = f_2(v'_2)$ for some function f_1 and f_2.

5 DTSN - Distributed Transport for Sensor Networks

DTSN [15] is a reliable transport protocol developed for sensor networks where intermediate nodes between the source and the destination of a data flow cache data packets in a probabilistic manner such that they can retransmit them upon request. The main advantages of DTSN compared to a transport protocol that uses a fully end-to-end retransmission mechanism is that it allows intermediate nodes to cache and retransmit data packets, hence, the average number of hops a retransmitted data packet must travel is smaller than the length of the route between the source and the destination. Intermediate nodes do not store all packets but only store packets with some probability p, which makes it be more efficient. Note that in the case of a fully end-to-end reliability mechanism, where only the source is allowed to retransmit lost data packets, retransmitted data packets always travel through the entire route from the source to the destination. Thus, DTSN improves the energy efficiency of the network compared to a transport protocol that uses a fully end-to-end retransmission mechanism.

DTSN uses special packets to control caching and retransmissions. More specifically, there are three types of such control packets: Explicit Acknowledgement Requests (EARs), Positive Acknowledgements (ACKs), and Negative Acknowledgements ($NACK$s). The source sends an EAR packet after the transmission of a certain number of data packets, or when its output buffer

becomes full, or when the application has not requested the transmission of any data during a predefined timeout period or due to the expiration of the *EAR* timer (*EAR_timer*).

The activity timer and the EAR timer are launched by the source for ensuring that a session will finish in a finite period of time. The activity timer is launched when the source starts to handle the first data packet in a session, and it is reset when a new packet is stored, or when an ACK or a NACK has been handled by the source. When the activity timer has expired, depending on the number of unconfirmed data packets, the session will be terminated or reset. The EAR timer is launched whenever an EAR packet or a data packet with the EAR bit set is sent.

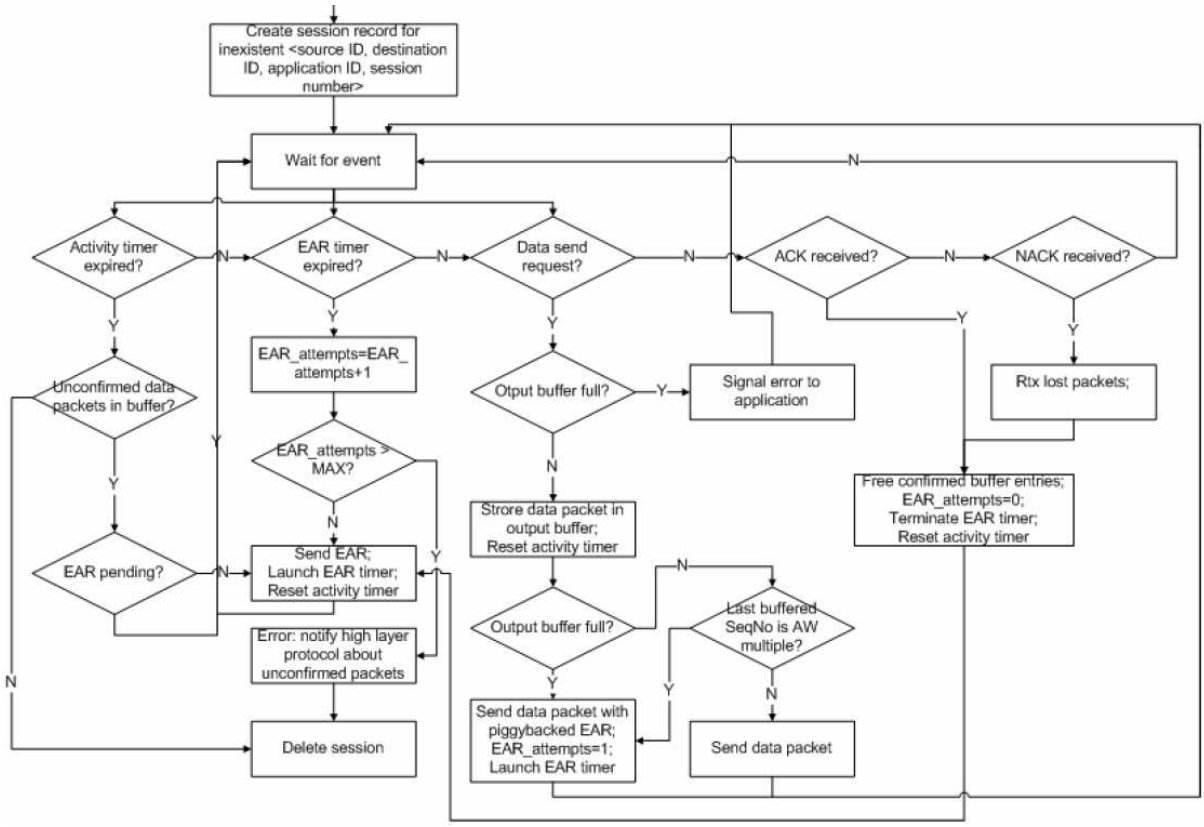

Figure 1: *The behavior of the source node in DTSN, taken from [15].*

An *EAR* may take the form of a bit flag piggybacked on the last data packet or an independent control packet. An *EAR* is also sent by an intermediate node or the source after retransmission of a series of data packets, piggybacked on the last retransmitted data packet [15]. Upon receipt of an *EAR* packet the destination sends an *ACK* or a *NACK* packet, depending on the existence of gaps in the received data packet stream. An *ACK* refers to a data packet sequence number n, and it should be interpreted such that all data packets with sequence number smaller than or equal to n were received by the destination. A *NACK* refers to a base sequence number n and it also contains a bitmap, in which each bit represents a different sequence number starting from the base sequence number n. A *NACK* should be interpreted such that all data packets with sequence number smaller than or equal to n were received by the destination and the data packets corresponding to the set bits in the bitmap are missing.

Within a session, data packets are sequentially numbered. The Acknowledgement Window (*AW*) is defined as the number of data packets that the source transmits before generating and sending an *EAR*. The output buffer at the sender works as a sliding window, which can span more than one *AW*. Its size depends on the specific scenario, namely on the memory constraints of individual nodes.

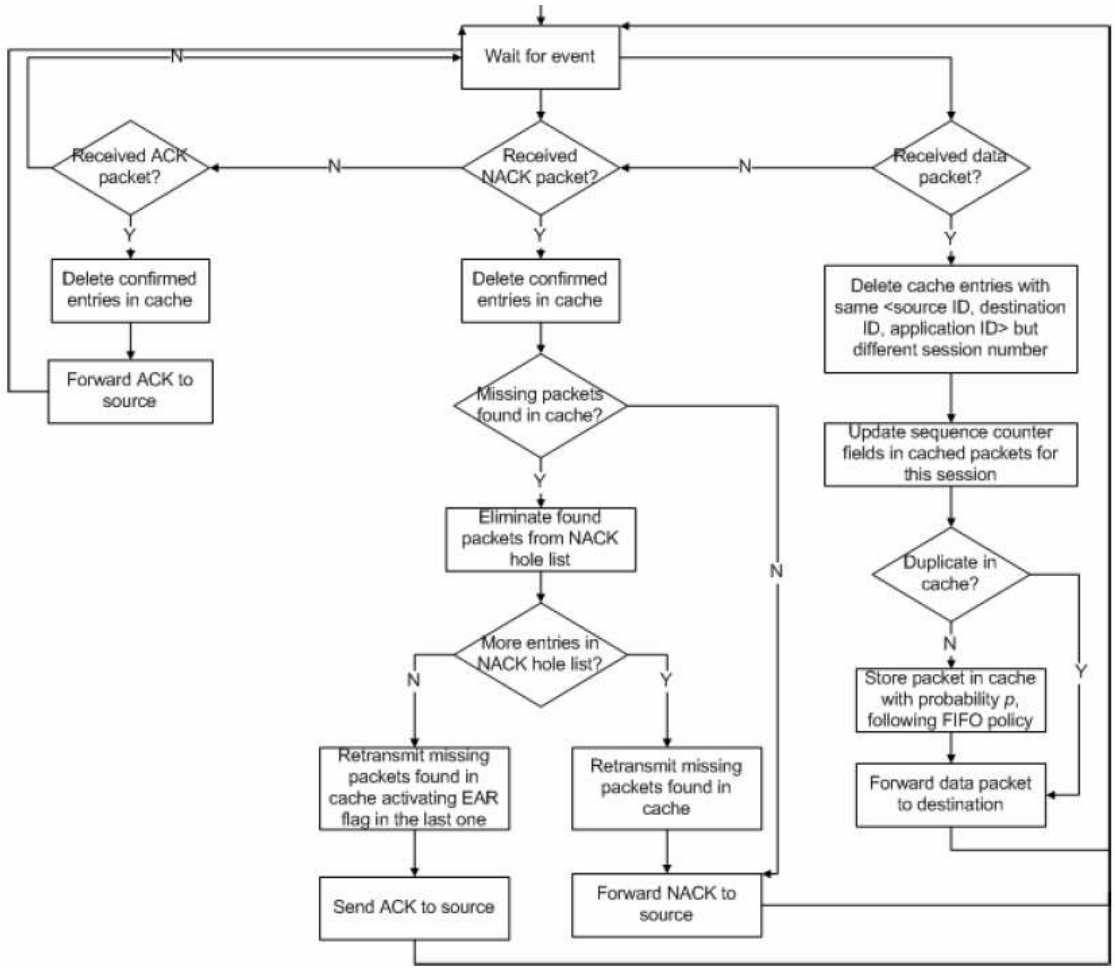

Figure 2: *The behavior of the intermediate node in DTSN, taken from [15].*

In DTSN, besides the source, intermediate nodes also process ACK and $NACK$ packets. When an ACK packet with sequence number n is received by an intermediate node, it deletes all data packets with sequence number smaller than or equal to n from its cache and passes the ACK packet on to the next node on the route towards the source. When a $NACK$ packet with base sequence number n is received by an intermediate node, it deletes all data packets with sequence number smaller than or equal to n from its cache and, in addition, it retransmits those missing data packets that are indicated in the $NACK$ packet and stored in the cache of the intermediate node. The bits that correspond to the retransmitted data packets are cleared in the $NACK$ packet, which is then passed on to the next node on the route towards the source. If all bits are cleared in the $NACK$, then the $NACK$ packet essentially becomes an ACK referring to the base sequence number, and it is processed accordingly. In addition, the intermediate node sets the EAR flag in the last retransmitted data packet. The source manages its cache and retransmissions in the same way as the intermediate nodes, without passing on any ACK and $NACK$ packets.

Security issues in DTSN. Upon receiving an ACK packet, intermediate nodes delete from their cache the stored messages whose sequence number is less than or equal to the sequence number in the ACK packet, because the intermediate nodes believe that acknowledged packets have been delivered successfully. Therefore, an attacker may cause permanent loss of some data packets by forging or altering ACK packets. This may put the reliability service provided by the protocol in danger. Moreover, an attacker can trigger unnecessary retransmission of the corresponding data

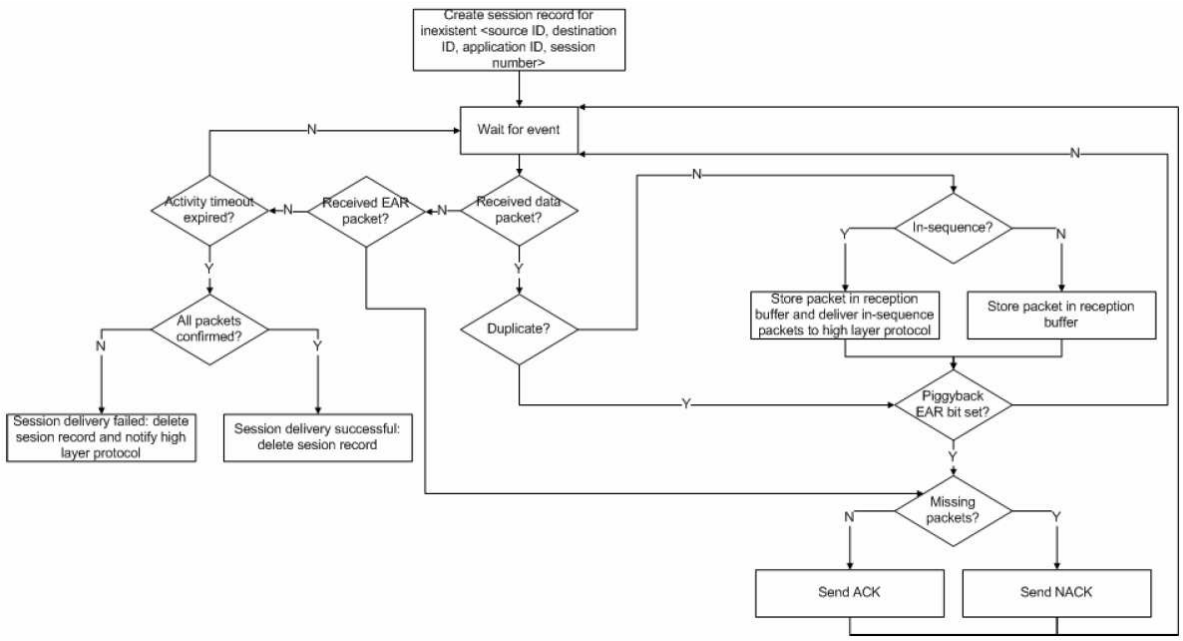

Figure 3: *The behavior of the destination node in DTSN, taken from [15].*

packets by either setting bits in the bit map of the *NACK* packets or forging/altering *NACK* packets. Any unnecessary retransmission can lead to energy consumption and interference. Note that, unnecessary retransmissions do not directly harm the reliability, but it is clear that such inefficiency is still undesirable.

The destination sends *ACK* or *NACK* packets upon reception of an *EAR*. Therefore, attacks aiming at replaying or forging *EAR* information, where the attacker always sets the *EAR* flag to 0 or 1, can have a harmful effect. Always setting the *EAR* flag to 0 prevents the destination from sending an *ACK* or *NACK* packet, while always setting it to 1 forces the destination send control packets unnecessarily.

5.1 DTSN in $\boldsymbol{crypt}^{prob}_{time}$

We assume the network topology $S{-}I{-}D$, where " $-$ " represents a bi-directional link, while S, I, D denote the source, an intermediate node, and the destination node, respectively. We also include the presence of the application that uses DTSN and SDTP, because it sends packet delivery requests to the source, and it receives delivered packets. In the rest of the paper we refer to the application as the *upper layer*.

Note that the attack scenarios which can be found and proved in this topology is also valid in other topologies including more intermediate nodes. Moreover, we assume that each node has three cache entries, denoted by e^s_k, e^i_k and e^d_k, $1{\leq}k{\leq}\ 3$. For brevity we let e^s_{1-3} range over e^s from index 1 to 3, and the same is true for e^i_{1-3} and e^d_{1-3}. We define symmetric channels between the upper layer and the source, c_{sup}; the upper layer and the destination, c_{dup}; the source and intermedate node, c_{si}; the intermediate node and the destination c_{id}. Moreover, we define additional channels c_{error} and $c_{sessionEND}$ for sending and receiving error and session-end signals.

We define $crypt^{prob}_{time}$ processes *upLayer*, *Src*, *Int*, *Dst* for specifying the behavior of the upper layer, the source, intermediate, and destination nodes. The DTSN protocol for the given topology is specified by the parallel composition of these four processes.

The specification of the DTSN protocol:

$Prot(params) \stackrel{def}{=}$
$\quad let\ (e^s_1,\ e^s_2\ ,\ e^s_3,\ e^i_1,\ e^i_2,\ e^i_3,\ e^d_1,\ e^d_2\ ,\ e^d_3,\ cntsq) = (E,\ E,\ E,\ E,\ E,\ E,\ E,\ E,\ E,\ 1)$
$\quad in\ INITDTSN();$
$INITDTSN() \stackrel{def}{=} \overline{c_{sup}}\langle cntsq\rangle.DTSN(params)$
$DTSN(params) \stackrel{def}{=}$
$\quad upLayer(incr(cntsq)) \mid initSrc(\text{s, d, apID, } e^s_{1-3}, \text{ sID, earAtmp}) \mid$
$\quad Int(e^i_{1-3}) \mid Dst(e^d_{1-3}, \text{ ackNbr, nackNbr, toRTX1, nxtsq});$

We refer to the tuple of parameters (*cntsq, s, d, apID,* e^i_{1-3}*, sID, earAtmp,* e^i_{1-3}*,* e^d_{1-3}*, ackNbr, nackNbr, toRTX1, nxtsq*) by (*params*). The process *Prot(params)* describes DTSN with variable initializations. The *let* construct is used to initialize the value of the cache entries to E, and the current sequence number to 1. The unique name E is used to represent the empty content. In the following, we give a brief overwiew of the main processes in our specification. Each main process is composed of additional sub-processes, which we skip discussing here. The processes are recursively invoked in a way to model replication.

We introduce two clock variables: x^{act}_c for the activity timer, and x^{ear}_c for the ear timer. According to the specification of the DTSN protocol [15], to model timeout we make use of the clock invariant defined on the process *Src*. The initial state of DTSN for the given topology is specified as the process $\parallel x^{act}_c, x^{ear}_c \parallel$ *Prot(params)*, which simply resets the timers at the beginning. We define the time amount of the activity and ear timers by T_{act} and T_{ear}, respectively. We assume that the activity timer is launched after the upper layer has sent the first request for the source, which is specified in $INITDTSN()$.

In process $INITDTSN()$, first of all, the request for sending the first packet with sequence number *cntsq* is sent. Then, the next request, $cntsq + 1$, is enqueued in $upLayer(incr(cntsq))$. The parameters of process *Src* are the IDs of the source and the destination; the application ID; the three cache entries, the session ID; and the latest number of EAR attempts. Process *Int* has the content of the three cache entries as parameter. Process *Dst* includes the *cache entries*; the *ACK/NACK numbers* for composing acknowlegement messages; the *packet to be re-transmitted* and the *next expected packet*.

In the following, we give a brief overwiew of each process *upLayer*, *Src*, *Int*, *Dst*, which are the main processes in our specification. Each main process is composed of additional sub-processes. For shortening the code we present the syntax sugar $upLayer(decr(cntsq))$ as the shorthand of process *let cntsq = decr(cntsq) in* $upLayer(cntsq)$. Intuitively, this process says that *upLayer* has the parameter which is $cntsq - 1$. Similarly, process $upLayer(incr(cntsq))$ gets the argument $cntsq + 1$.

Process that models the behavior of the upper layer:

$upLayer(\text{cntsq}) \stackrel{def}{=}$
$\quad c_{sup}(= PRVOK).hndlePck(\text{cntsq})\ [\ ]\ c_{error}(= ERROR).upLayer(decr(cntsq))$
$\quad [\ ]\ c_{sessionEND}(= SEND).\mathbf{nil}\ [\ ]\ c_{dup}(x_{pck}).upLayer(cntsq);$

Process *upLayer* has the current sequence number as parameter, denoted by *cntsq*, which is used to keep track of the next or previous sequence number. Process $c_{sup}(= PRVOK).hndlePck(\text{cntsq})$ first waits for a feedback from *Src* indicating that the source has received and handled the request regarding *cntsq*. Input process $c_{sup}(= PRVOK)$ checks if the received message is equal to the constant *PRVOK*, meaning that the source has already handled the latest request, hence, the upper layer can go on with the next request, this is described in process $hndlePck(cntsq)$.

$hndlePck(\text{cntsq}) \stackrel{def}{=}$
$\quad [cntsq \leq mxSQ]\ (\overline{c_{sup}}\langle cntsq\rangle.upLayer(incr(cntsq)))$
$\quad else\ upLayer(\text{cntsq});$

Intuitively, in case *cntsq* is less than or equal to the maximal sequence number, denoted by *mxSQ*, the request for delivering packet *cntsq* is initiated by sending *cntsq* via channel c_{sup}, followed by the recursive invocation of *upLayer* with the next sequence number. Otherwise, *upLayer* is invoked again with *cntsq* unchanged.

Whenever the upper layer receives an error message, which is specified by the input of the constant *ERROR* on channel c_{error}, process *upLayer* is invoked with $cntsq - 1$. Namely, this addresses the scenario when the source cannot handle the latest delivery request, hence, the upper layer repeats this request. Whenever, a session-end signal, which is defined as the constant *SEND*, is received the upper layer terminates its operation. Finally, when the upper layer receives a data packet, on channel c_{dup}, delivered by the destination it continues its operation.

The source handling the activity timer expiration:

InitSrc(s, d, apID, e^s_{1-3}, sID, earAtmp) $\stackrel{def}{=}$

1. $c_{sup}(x_{sq}).($
   $\{\ x_c^{act} \leq T_{act}\ \} \triangleright (x_c^{act} \leq T_{act}) \hookrightarrow$ *initFwdDt*(s, d, apID, e^s_{1-3}, sID, x_{sq})
   $[\ ]\ \{x_c^{act} \leq T_{act}\} \triangleright (x_c^{act} \leq T_{act}) \hookrightarrow$ *initRcvACKS*(s, d, apID, e^s_{1-3}, sID, earAtmp)
   $[\ ]\ \{x_c^{act} \leq T_{act}\} \triangleright (x_c^{act} \leq T_{act}) \hookrightarrow$ *initRcvNACKS*(s, d, apID, e^s_{1-3}, sID, earAtmp)
)
2. $[\ ]\ \{x_c^{act} \leq T_{act}\} \triangleright (x_c^{act} \leq T_{act}) \hookrightarrow c_{sessionEND}(= SEND)$.**nil**
3. $[\ ]\ (x_c^{act} \geq T_{act}) \hookrightarrow$ *actTimeOut*;

The process *initSrc* specifies the source node starting with the first packet delivery, when the source does not launch the EAR timer yet, but only the ACT (activity) timer. The sub-process *actTimeOut* describes the behavior of the protocol when the ACT timer has expired. Similarly, for the EAR timer we define the process *earTimeOut*.

The three choice options represent the "wait for event" activity of the source. The choice is resolved after the corresponding event occurs. Each choice option represents a scenario: The last (third) option in point 3 describes the case when the activity timer has elapsed. According to the definition of the source in DTSN and SDTP (Figure 1), the source checks the expiration of the *EAR* and the *ACT* timers, only when it returned to the status *Wait for event*. This means that within a branch corresponding to the received message (a data, an *ACK*, a *NACK*), timer expirations do not interrupt the behavior of the source immediately. Moreover, whenever the source steps into one branch, in case of DTSN, the timers will be reset within the branch, before returning to the status *waiting for event*. The process $(x_c^{act} \geq T_{act}) \hookrightarrow$ *actTimeOut* says that when the *ACT* timer has expired the protocol proceeds with the process *actTimeOut*, which describes the defined behavior of *Src* after timeout. The second choice (point 2.) is for the case when the session is terminated, which happens when the constant *SEND* has been sent on the private channel $c_{sessionEnd}$ by the source [15]. Each node is defined such that it waits for *SEND* by the construct $c_{sessionEND}(= SEND)$, where $=$*SEND* is used to check if the received data is equal to *SEND*. This construct is basically the abbreviation of the process $c_{sessionEND}(x_{SEND})$. $[x_{SEND} = SEND]$. After receiving the session end signal each node terminates its operation.

We assume that the session termination cannot be interrupted by the timeouts, basically, it can be seen as an atomic action. When the first option has been chosen, it means that in that step the source received the delivery request from the upper layer, and that no session end or timeouts happen during this input action.

Specifically, in *initFwdDt*(s, d, apID, e^s_{1-3}, sID, sq), after checking that the buffer is not full, the source stores the packet in a cache entry, then, after reseting the ACT timer, it proceeds with checking the saturation of the buffer.

The source stores the received packet and resets the ACT timer:

initFwdDt(s, d, apID, e^s_{1-3}, sID, sq) $\stackrel{def}{=}$

. . . . . .
let $e_i^s = (s, d, apID, sID, sq)$ *in*
$\| \; x_c^{act} \; \|$ *checkBuffandAW*(s, d, apID, e_{1-3}^s, sID, sq, earAtmp)

First, the packet is stored in one of the cache entries, $i \in \{1, 2, 3\}$. The process $\| \; x_c^{act} \; \| \; \{ \; x_c^{act} \leq T_{act} \; \} \triangleright (x_c^{act} \leq T_{act})$ *checkBuffandAW* resets the ACT timer and continues the behavior of the source by checking whether the buffer is full and if *sq* is AW multiple. For simplicity, we assume AW be 2, hence, we need only to check if *sq* is equal to 2 or 4, because we consider at most 4 packets in a session. After that, the source launches the EAR timer and sets the EAR attempt to 1, followed by repeating its operation (i.e., it waits for an event).

Checking buffer full and launches the EAR timer:

checkBuffandAW(s, d, apID, e_{1-3}^s, sID, sq, earAtmp) $\stackrel{def}{=}$
. . . . . .
$\overline{c_{si}}\langle(s, d, apID, sID, sq, ear, rtx)\rangle$. *let* $earAtmp = 1$ *in* $\| \; x_c^{ear} \; \|$ *Src*(s, d, apID, e_{1-3}^s, sID, earAtmp)

We only discuss such portion of process *checkBuffandAW* which is related to the timing issue such as lauching the EAR timer. First, the packet with the sequence number *sq* and ear bit set to 1, is sent to I. Then, after setting the EAR attempt to 1, it launches the EAR timer. We note that the appearance of x_c^{act} in $\{x_c^{act} \leq T_{act}, \; x_c^{ear} \leq T_{ear}\}$ does not means that it has been reset/launched again, but only that the time invariant for the clock x_c^{act} is carried forward.

The processes *initRcvACKS* and *initRcvNACKS* are activated when an ACK/NACK message is received by the source node at the very beginning of the protocol. At the beginning, no packet is stored, hence, these two processes are very simple: basically, whatever is input on the channels c_{siACK}, c_{siNACK} they proceed to the process $\{x_c^{act} \leq T_{act}\} \triangleright$ *Src*(*parameters*).

Process *Src*(s, d, apID, e_{1-3}^s, sID, earAtmp) is a bit differ from *initSrc*(s, d, apID, e_{1-3}^s, sID, earAtmp) in that the *EAR* timer will be launched in it. *Src* considers the case when the source has already stored some packets, and is located within the process *checkBuffandAW* after the source sent the current data packet, and it is located in *initRcvACKS*, and *initRcvNACKS* after the source has finished deleting its buffer and re-transmitting the required packets, according to the received *ACK* and *NACK* packets.

The source's activity after it has already stored packets :

Src(s, d, apID, e_{1-3}^s, sID, earAtmp) $\stackrel{def}{=}$
5. $c_{sup}(x_{sq})$. $\{ \; x_c^{act} \leq T_{act}, \; x_c^{ear} \leq T_{ear} \; \} \triangleright (x_c^{act} \leq T_{act}, \; x_c^{ear} \leq T_{ear}) \hookrightarrow$
fwdDt(s, d, apID, e_{1-3}^s, sID, x_{sq})
[ ] $c_{siACK}(x_{ack})$. $\{ \; x_c^{act} \leq T_{act}, \; x_c^{ear} \leq T_{ear} \; \} \triangleright (x_c^{act} \leq T_{act}, \; x_c^{ear} \leq T_{ear}) \hookrightarrow$
rcvACKS(s, d, apID, e_{1-3}^s, sID, earAtmp, x_{ack})
[ ] $c_{siNACK}(x_{nack})$. $\{ \; x_c^{act} \leq T_{act}, \; x_c^{ear} \leq T_{ear} \; \} \triangleright (x_c^{act} \leq T_{act}, \; x_c^{ear} \leq T_{ear}) \hookrightarrow$
rcvNACKS(s, d, apID, e_{1-3}^s, sID, earAtmp, x_{nack})
6. [ ] $\{ \; x_c^{act} \leq T_{act}, \; x_c^{ear} \leq T_{ear} \; \} \triangleright (x_c^{act} \leq T_{act}, \; x_c^{ear} \leq T_{ear}) \hookrightarrow c_{sessionEND}(= SEND)$.**nil**
7. [ ] $(x_c^{act} \geq T_{act}) \hookrightarrow$ *actTimeOut*
8. [ ] $(x_c^{ear} \geq T_{ear}) \hookrightarrow$ *earTimeOut*;

Beside the *ACT* timer, points 5-8 also include the reseting of the *EAR* timer, hence, the clock invariant $\{x_c^{act} \leq T_{act}\}$ in points 1-3 is extended with $\{x_c^{ear} \leq T_{ear}\}$, and the action guard becomes $(x_c^{act} \leq T_{act}, \; x_c^{ear} \leq T_{ear})$. In addition, the processes *fwdDt*, *rcvACKS* and *rcvNACKS* differ from *initFwdDt*, *initRcvACKS*, and *initRcvNACKS*, such that in the first three processes, the source has to perform some searching steps. Finally, in process *Src*, the timeout of the *EAR* timer should be taken into account (in point 8). The process *Src* is located at the end of the processes *fwdDt*, *rcvACKS* and *rcvNACKS* to model the recursive behavior of the source, until the session end.

The source handles activity timer expiration:

$actTimeOut \stackrel{def}{=}$
$[nbrUnConfirmed = 0]\ \overline{c_{sessionEND}}\,\langle SEND\rangle.\mathbf{nil}$
$[\,]\ [nbrUnConfirmed > 0]\ \overline{c_{si}}\,\langle EAR\rangle.$
$\quad \parallel x_c^{act},\ x_c^{ear} \parallel Src$(s, d, apID, e^s_{1-3}, sID, earAtmp);

In *actTimeOut*, according the definition of DTSN, if there is not any unconfirmed packet in the buffer, the session is terminated. Otherwise, the *EAR* packet is sent on channel c_{si}, then, the *ACT* and *EAR* timers are reset, followed by waiting for an event after invoking recursively the process *Src*.

Process *earTimeOut* defines the behavior of the source in case of EAR timeout, namely, the EAR attempt is increased. Then, if the EAR attempt exceeds the MAX nnumber, the session is terminates, otherwise, the ACT and EAR timers are reset, and it waits for events.

the source handles EAR timer expiration:

$earTimeOut \stackrel{def}{=}$
let earAtmp = incr(earAtmp) in
(
$[earAtmp > earMAX]\ \overline{c_{sessionEND}}\,\langle SEND\rangle.\mathbf{nil}$
$[\,]$
$[earAtmp \leq earMAX]\ \overline{c_{si}}\,\langle EAR\rangle.$
$\parallel x_c^{act},\ x_c^{ear} \parallel Src$(s, d, apID, e^s_{1-3}, sID, earAtmp)
);

In process *earTimeOut*, the *EAR* attempts are increased. Then, if the *EAR* attempts exceed the MAX number, the session is terminates, otherwise, the *ACT* and *EAR* timers are reset, and it waits for events by invoking recursively the process *Src*.

In addition, we have to add the resetting of the timers at the source node after handling ACK/NACK packets. Hence, the processes *rcvACKS* and *rcvNACKS* are extended as follows:

$rcvACKS$(s, d, apID, e^s_{1-3}, sID, earAtmp) $\stackrel{def}{=}$
$c_{siACK}(x_{acknum}).hndleACK$(s, d, apID, e^s_{1-3}, sID, x_{acknum});

$hndleACK$(s, d, apID, e^s_{1-3}, sID, acknum) $\stackrel{def}{=}$
$[5(e^s_1) \leq acknum]\ checkE1$(s, d, apID, e^s_{1-3}, sID, acknum) *else*
$[5(e^s_2) \leq acknum]\ checkE2$(s, d, apID, e^s_{1-3}, sID, acknum) *else*
$[5(e^s_3) \leq acknum]\ checkE3$(s, d, apID, e^s_{1-3}, sID, acknum) *else*
/* Here we add the resetting of the two timers on process Src */
let (earAtmp = 0) in $\parallel x_c^{act},\ x_c^{ear} \parallel Src$(s, d, apID, e^s_{1-3}, sID, earAtmp);

Process *hndleACK* specifying how the source behaves after receiving an ACK message. It starts with checking if the packet in the first cache entry has a sequence number less than the received *acknum*. The construct $5(e^s_i)$ specifies the 5-th element of the packet stored in the i-th cache entry e^s_i, which is the sequence number. In case $(5(e^s_1) \leq acknum)$, the source deletes the content of e^s_1, and continues with checking the sequence number in the second cache entry. These are included in the process *checkE1*, where *E1* refers to the first cache entry. Otherwise, if $(5(e^s_1) > acknum)$ holds, then the source continues with the same steps for the second entry, and so on. The last two rows say that after examining every cache entry the source sets the EAR attempts to zero, and resets the EAR and ACT timers.

The processes *checkE1*, *checkE2* and *checkE3* starts with deleting the corresponding cache entry by using the process ***let*** $(e_i^s = E)$ ***in*** ..., which gives the special name E (i.e., "Empty") to e_i^s, then continues with the same steps as in *hndleACK*. For instance, the code of *checkE1* is as follows:

$checkE1$(s, d, apID, e_{1-3}^s, sID, acknum) $\stackrel{def}{=}$ *let* $(e_1^s = E)$ *in*
$[5(e_2^s) \leq acknum]$ *checkE2*(s, d, apID, e_{1-3}^s, sID, acknum) *else*
$[5(e_3^s) \leq acknum]$ *checkE3*(s, d, apID, e_{1-3}^s, sID, acknum) *else*
/* Here we add the resetting of the two timers on process Src */
let $(earAtmp = 0)$ *in* $\| x_c^{act}, x_c^{ear} \|$ *Src*(s, d, apID, e_{1-3}^s, sID, earAtmp);

According to the definition of DTSN [15], the activity timer expiration also affects the behavior of the destination. Upon activity timer expiration, the destination checks if all the packets has been received and confirmed, and in case yes the session terminates with success, otherwise, it terminates with error. We simplify the model, without lost of correctness, by not specify explicitly this behavior for the destination, but by the channel $c_{endSession}$. When activity timer expired the source sends session-end signal, which is then received by the destination (and intermediate node) on channel $c_{endSession}$ which followed by the special process **nil**. Intuitively, the end session command is given by the source which is then performed by the other processes. Recall that the attacker cannot interfere the information sent on $c_{endSession}$, because it is private. We can omit the specification of the time constructs in the rest processes, hence, avoiding the conflict of clock variables. Specifically, we extend the scope of clocks resets on *Src* to the rest processes, applying the scope extrusion rule for parallel composition:

$$(\|C\|\ A_t^1) \mid A_t^2 \equiv \|C\|\ (A_t^1 \mid A_t^2) \text{ if } C \cap fvc(A_t^2) = \emptyset$$

The denotation $\equiv$ refers to the notion of structural equivalence, which is well-known in process calculi. It is used to define when two processes are structurally equivalence. For example, $A_1 \mid A_2$ is equivalence to $A_2 \mid A_1$. The rule above says that if the set of clocks to be reset are not among the free clock variables of A_t^2, then $\|C\|$ can be extended. For the process $\|C\|\ (A_t^1 \mid A_t^2)$ we can define the corresponding location in timed automata.

Process that models the behavior of an intermediate node:

$Int(e_{1-3}^i) \stackrel{def}{=}$
$c_{si}((x_s, x_d, x_{apID}, x_{sID}, x_{sq}, x_{ear}, x_{rtx})).hndleDtI$(s, d, apID, sID, sq, ear, rtx, e_{1-3}^i)
[ ] $rcvACKI(e_{1-3}^i)$ [ ] $rcvNACKI(e_{1-3}^i)$ [ ] $c_{sessionEND}(= SEND)$.**nil**;

In process *Int* the intermediate node may receive a data packet on channel c_{si} and handles the received packet according to the definition of DTSN, it can receive and handle an ACK or NACK message, and it can terminate its operation when it gets the signal.

Process that models the behavior of the destination:

$Dst(e_{1-3}^d$, ackNbr, nackNbr, toRTX1, nxtsq) $\stackrel{def}{=}$
$c_{id}((x_s, x_d, x_{apID}, x_{sID}, x_{sq}, x_{ear}, x_{rtx})).hndleDtDst$ [ ] $c_{sessionEND}(= sEND)$.**nil**;

For the process *Dst*, the destination can either receive a data packet on channel c_{id} or receive a session end signal. In the first case, *Dst* proceeds with *hndleDtDst* in which the destination performs the verification steps and delivers the packet to the upper layer, or sending an ACK or a NACK.

In the following, we extend the description of the DTSN protocol in $crypt_{time}$ with a probabilistic choice. According to the definition of the DTSN protocol, the probabilistic choice is placed within process $Int(e_{i-3}^i)$, which is the specification of intermediate nodes. In particular, after receiving a packet an intermediate node stores the packet in its cache with probability p. To model this behavior we add the probabilistic choice construct in the sub-process *hndleDtI*, "reponsible" for handling a received data packet.

$$\begin{aligned}
&\mathit{hndleDtI}(\text{s, d, apID, sID, sq, ear, rtx, } e^i_{1-3}) \stackrel{def}{=}\\
&\quad [e^i_1 = (s,d,apID,sID,sq)]\ \overline{c_{id}}\langle s,d,apID,sID,bt,ear,rtx\rangle.\mathit{Int}(e^i_{1-3})\ \mathit{else}\\
&\quad [e^i_2 = (s,d,apID,sID,sq)]\ \overline{c_{id}}\langle s,d,apID,sID,bt,ear,rtx\rangle.\mathit{Int}(e^i_{1-3})\ \mathit{else}\\
&\quad [e^i_3 = (s,d,apID,sID,sq)]\ \overline{c_{id}}\langle s,d,apID,sID,bt,ear,rtx\rangle.\mathit{Int}(e^i_{1-3})\ \mathit{else}\\
&\quad \mathit{strAndFwI}(\text{s, d, apID, sID, sq, ear, rtx, } e^i_{1-3}) \oplus_p \mathit{FwI}(\text{s, d, apID, sID, sq, ear, rtx, } e^i_{1-3});
\end{aligned}$$

$$\mathit{FwI}(\text{s, d, apID, sID, sq, ear, rtx, } e^i_{1-3}) \stackrel{def}{=} \overline{c_{id}}\langle s,d,apID,sID,bt,ear,rtx\rangle.\mathit{Int}(e^i_{1-3});$$

we add the probabilistic choice

$$\mathit{strAndFwI}(\text{s, d, apID, sID, sq, ear, rtx, } e^i_{1-3}) \oplus_p \mathit{FwI}(\text{s, d, apID, sID, sq, ear, rtx, } e^i_{1-3}),$$

in which process *strAndFwI* that describes when the intermediate node stores the received packet, is chosen with probability p, and process *FwI* that specifies that the received packet is forwarded without storing, is selected with $1 - p$.

6 SDTP - A Secure Distributed Transport Protocol for WSNs

SDTP is a security extension of DTSN aiming at patching the security holes in DTSN. SDTP ensures that an intermediate node can verify if an acknowledgment or negative acknowledgment information has really been issued by the destination, if and only if the intermediate node actually has in its cache the data packet referred to by the ACK or NACK. Forged control information can propagate in the network, but only until it hits an intermediate node that cached the corresponding data packet; this node can detect the forgery and drop the forged control packet.

In particular, the security solution of SDTP works as follows [5]: each data packet is extended with an *ACK* MAC and a *NACK* MAC, which are computed over the whole packet with two different keys, an *ACK* key (K_{ACK}) and a *NACK* key (K_{NACK}). Both keys are known only to the source and the destination and are specific to the data packet; hence, these keys are referred to as per-packet keys.

When the destination receives a data packet, it can check the authenticity and integrity of each received data packet by verifying the two MAC values. Upon receipt of an *EAR* packet, the destination sends an *ACK* or a *NACK* packet, depending on the gaps in the received data buffer. If the destination sends an *ACK* referring to a data packet with sequence number n, the destination reveals (included in the *ACK* packet) the corresponding *ACK* key; similarly, when it wants to signal that this data packet is missing, the destination reveals the corresponding *NACK* key by including it in the *NACK* packet. Any intermediate node that stores the packets in question can verify if the *ACK* or *NACK* message it receives is authentic by checking if the appropriate MAC in the stored data packet verifies correctly with the *ACK* key included in the *ACK* packet. In case of successful verification, the intermediate node deletes the corresponding data packets (whose sequence number is smaller than or equal to n) from its cache.

When an ACK packet is received by an intermediate node or the source, the node first checks if it has the corresponding data packet. If not, then the ACK packet is simply passed on to the next node towards the source. Otherwise, the node uses the ACK key obtained from the ACK packet to verify the ACK MAC value in the data packet. If this verification is successful, then the data packet can be deleted from the cache, and the ACK packet is passed on to the next node towards the source. If the verification of the MAC is not successful, then the ACK packet is silently dropped.

When a NACK packet is received by an intermediate node or the source, the node processes the acknowledgement part of the NACK packet as described above. In addition, it also checks if it has any of the data packets that correspond to the set bits in the bitmap of the NACK packet. If it does not have any of those data packets, it passes on the NACK without modification. Otherwise, for each data packet that it has and that is marked as missing in the NACK packet, it verifies

the NACK MAC of the data packet with the corresponding NACK key obtained from the NACK packet. If this verification is successful, then the data packet is scheduled for re-transmission, the corresponding bit in the NACK packet is cleared, and the NACK key is removed from the NACK packet. After these modifications, the NACK packet is passed on to the next node towards the source.

The ACK and $NACK$ key generation and management in SDTP is as follows: The source and the destination share a secret which we call the session master key, and we denote it by K. From this, both the source and destination derive an ACK master key K_{ACK} and a NACK master key K_{NACK} for a given session as follows:

$$K_{ACK} = \text{PRF}(K; \text{“ACK master key”; SessionID})$$
$$K_{NACK} = \text{PRF}(K; \text{“NACK master key”; SessionID})$$

where PRF is a pseudo-random function [7], and SessionID is a session identifier.

SDTP assumes a pre-established shared secret value, such as a node key shared by the node and the base station, which can be configured manually in the node before its deployment. Denoting the shared secret by S, the session master key K is then derived as follows:

$$K = \text{PRF}(S; \text{“session master key”; SessionID})$$

The ACK key $K_{ACK}^{(n)}$ and the NACK key $K_{NACK}^{(n)}$ for the n-th packet (i.e., whose sequence number is n) are computed as follows:

$$K_{ACK}^{(n)} = \text{PRF}(K_{ACK}; \text{“per packet ACK key”}; n)$$
$$K_{NACK}^{(n)} = \text{PRF}(K_{NACK}; \text{“per packet NACK key”}; n)$$

Note that both the source and the destination can compute all these keys as they both possess the session master key K. Moreover, PRF is a one-way function, therefore, when the ACK and NACK keys are revealed, the master keys cannot be computed from them, and consequently, as yet unrevealed ACK and NACK keys remain secrets too.

Security issues in SDTP. The rationality behind this security solution is that the shared secret S is never leaked, and hence, only the source and the destination can produce the right ACK and NACK master keys and per-packet keys. Since the source never reveals these keys, the intermediate node can be sure that the control information has been sent by the destination. In addition, because the perpacket keys are computed by a one-way function, when the ACK and NACK keys are revealed, the master keys cannot be computed from them; hence, the yet unrevealed ACK and NACK keys cannot be derived. These issues give the protocol designers an impression that SDTP is secure, however, we will formally prove that SDTP is still vulnerable and showing a tricky attack against it.

6.1 SDTP in $\boldsymbol{crypt}_{time}^{prob}$

The non-cryptographic parts of SDTP including the timing and probabilistic elements are specified in the same way as in case of the DTSN protocol. Hence, I focus on the cryptographic parts of SDTP. To model the cryptographic primitives and operations in SDTP, I add the following equations into the set of equational theories:

$$\textit{Functions}\colon K(n, ACK);\ K(n, NACK);\ mac(t, K(n, ACK));\ mac(t, K(n, NACK));$$

$$\begin{aligned}\textit{Equations}\colon\ & checkmac(mac(t, K(n, ACK)), K(n, ACK)) = ok;\\ & checkmac(mac(t, K(n, NACK)), K(n, NACK)) = ok.\end{aligned}$$

where functions $K(n, ACK)$ and $K(n, NACK)$ specify the ACK and $NACK$ per-packet keys corresponding to the packet with sequence number n. The functions $mac(t, K(n, ACK))$ and $mac(t, K(n, NACK))$ compute the MAC on the message t, using the ACK and the $NACK$ keys

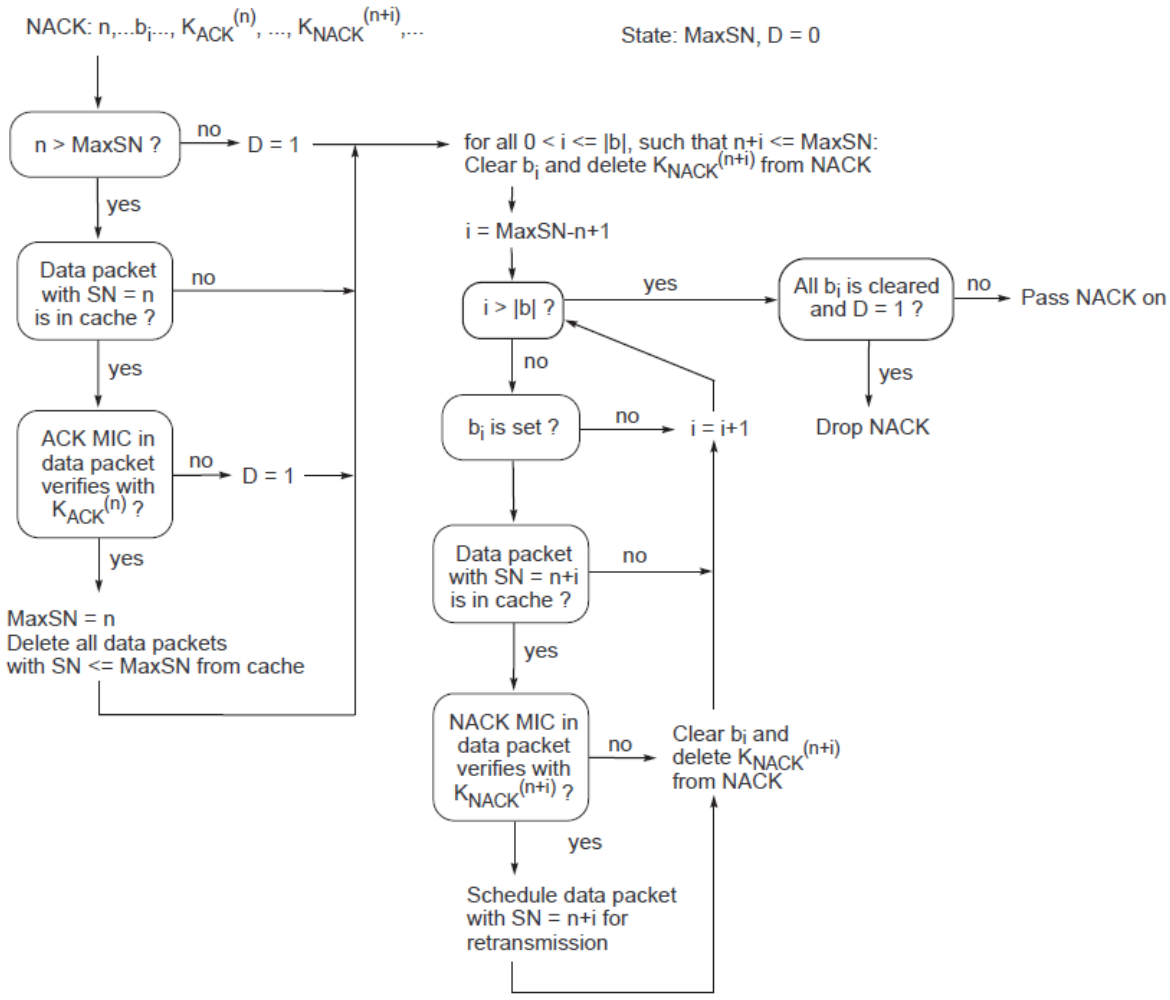

Figure 4: *The NACK in SDTP, taken from [5].*

for the n-th packet. In order to simplify the modelling procedure, without violating the correctness of SDTP, I make an abstraction of the key hierarchy given in [5], where the per-packet keys are computed with a one-way function based on the shared secret unknown to the attacker. Instead, I assume that $K(n, ACK)$ and $K(n, NACK)$ cannot be generated (but can be intercepted) by the attackers. The attackers can only generate keys that are differ from these keys. With this, I model the fact that the shared secret will never be revealed during the protocol.

One most important change, compared to DTSN, resulted from the specification of the SDTP is that the destination node includes ACK and NACK keys in the ACK and NACK messages. To model the SDTP protocol we extend the specification of the DTSN protocol in the following way. First, the source node extends each packet with an ACK MAC and a NACK MAC, then sends it to node I, which is accomplished by the following code part in the process *Src*. Let us consider, for example, the process *checkBuffandAW* discussed above.

Checking buffer full and launches the EAR timer:

checkBuffandAW(s, d, apID, e^s_{1-3}, sID, sq, earAtmp) $\stackrel{def}{=}$

. . . . . .

let ear=val1 in let rtx=val2 in let earAtmp=val3 in
let Kack = K(sq, ACK) in let Knack = K(sq, NACK) in
let ACKMAC = MAC((s, d, apID, sID, sq, ear, rtx), Kack) in

let NACKMAC = MAC((s, d, apID, sID, sq, ear, rtx), Knack) in
$\overline{c_{si}}\langle$(*s, d, apID, sID, sq, ear, rtx, ACKMAC, NACKMAC*)$\rangle$.*let earAtmp* = 1 *in*
$\parallel x_c^{ear} \parallel$ *Src*(s, d, apID, e^s_{1-3}, sID, earAtmp);

In the first row the variables *ear*, *rtx*, and *earAtmp* are given some values *val1*, *val2* and *val3*, respectively. In the second row the ACK/NACK keys Kack and Knack are generated, while in the third and fourth rows the ACK/NACK MACs are computed using the generated ACK/NACK keys. Finally, the whole packet is output on channel c_{si} and the process *Src* is recursively invoked.

In addition, we have to extend the $crypt^{prob}_{time}$ specification of DSTN with the verification of ACK MACs and NACK MACs when the source receives ACK and NACK packets. Formally, we extend the processes

rcvACKS(s, d, apID, e^s_{1-3}, sID, earAtmp) $\stackrel{def}{=}$
c_{siACK}(acknum, **ackkey**, **nackkey**).
hndleACK(s, d, apID, e^s_{1-3}, sID, acknum, **ackkey**, **nackkey**);

such that the expected data on channel c_{siACK} is extended by *ackkey*, *nackkey* that represent per-packet ACK and NACK keys, which are also included as the parameters of process *hndleACK* and its sub-processes. We extend the specification of *hndleACK* with the verification of the stored ACK MAC using the keys included in the received ACK packets. This is modelled by the *if* construct in $crypt^{prob}_{time}$: $[checkmac(6(e^s_i), ackkey) = ok]$. In particular, $checkmac(6(e^s_i), ackkey)$ is the verification of the received ACK MAC, which is stored in the 6-th place in the cache entry e^s_i, denoted and defined by the function $6(e^s_i)$. The same verification is applied in the sub-processes.

When a NACK packet has been received the SDTP protocol includes verification of ACK MAC and NACK MACs. The structure of the NACK packet compared to DTSN case is extended with an ACK key (if any) and some NACK keys depending on the number of bits in the NACK packet. Hence, the expected data on channel c_{siNACK} is extended with the ackkey and nackkey parameters, for instance, instead of c_{siNACK}(*acknum*, *b1*) we have c_{siNACK}(*acknum*, *b1*, *ackkey*, *nackkey1*). Namely, the verification part $[5(e^s_i) \leq acknum]$, which examines if the 5-th element of the entry e^s_i (i.e. the stored sequence number *sq*) is less or equal to the received *acknum*, is extended with the verification of the ACK/NACK MACs $[checkmac(6(e^s_i), ackkey) = ok]$ $[checkmac(7(e^s_i), nackkey) = ok]$ for each $i \in \{1, 2, 3\}$.

Now we turn to modify the process $Int(e^i_{1-3})$ according to the definition of the SDTP protocol. Beside the parameters already defined in case of DTSN, the process *Int* in SDTP also includes *ackmac* and *nackmac* parameters. Each cache entry at intermediate nodes stores the packets that contains an ACK MAC and NACK MAC at the 6-th and 7-th places, respectively.

In process *rcvACKI*, which describes the behavior of node *I* after receiving an ACK, the ACK message format on channel c_{idACK} is extended with *ackkey*, (*acknum*, *ackkey*), and process *hndleACKI* also contains *ackkey* as its parameter.

rcvACKI(e^i_{1-3}) $\stackrel{def}{=}$
c_{idACK}(acknum, ackkey).hndleACKI(s, d, apID, e^i_{1-3}, sID, acknum, ackkey);

The specifications of processes *hndleACKI*, and the sub-processes within *hndleACKI* are modified similarly as in case of *hndleACK* and its sub-processes in DTSN.

We define a process, called *rcvNACKI*, which specifies the behavior when node *I* receives a NACK message. In this process the expected messsage format on channel c_{idNACK} includes the ack number and the packets need to be retransmitted, and the corresponding ACK/NACK keys, e.g., c_{idNACK}(*xacknum, xb1, xackkey, xnackkey*).

Finally, the process *Dst* for the destination node is modified such that the expected message on channel c_{id}, is extended with the ACK and NACK MACs. Then, *Dst* performs the verification operations on ACK/NACK MACs.

7 Security Analysis of DTSN and SDTP using $crypt^{prob}_{time}$

In the next two subsections, we formally prove the insecurity of the DTSN and SDTP protocols using the weak prob-timed bisimilarity defined in $crypt^{prob}_{time}$, and we also specify their behavior using the syntax of $crypt^{prob}_{time}$.

In our formal proofs we apply the proof technique that is usual in process algebras, such as the spi [1] and the applied π calculi. Namely, we define an ideal version of the protocol run, in which we specify the ideal/secure operation of the real protocol. This ideal operation, for example, can be that honest nodes always know what is the correct message they should receive/send, and always follows the protocol correctly, despite the presence of attackers. Then, we examine whether the real and the ideal versions, besides the same attacker(s), are probabilistic timed bisimilar.

Definition 12. *Let the processes Prot() and $Prot^{ideal}()$ specify the real and ideal versions of some protocol Prot, respectively. We say that Prot is secure (up to the strictness of the ideal version) if Prot() and $Prot^{ideal}()$ are probabilistic timed bisimilar: Prot() $\approx_{pt}$ $Prot^{ideal}()$.*

The strictness of the security requirement, which we expect a protocol to fullfil, depends on how "perfectly" we specify the ideal version. Intuitively, Definition 12 says that *Prot* is secure if the attackers, who can observe the output messages, cannot distinguish the operation of the two instances.

7.1 Security Analysis of the DTSN protocol

The security properties we want to check in case of DTSN protocol is that how secure it is against the manipulation of control and data packets. In particular, can the manipulation of packets results in unintentional closing of a session or preventing DTSN from achieving its design goal. Due to lack of security mechanisms, DTSN is vulnerable to the manipulation of control packets, where the attacker can modify the base number in ACK packets causing that the stored packets are deleted from the cache although they should not be (by increasing the base number), or causing unnecessary storage of the already delivered packets (by decreasing the base number). In this section we demonsrate how to formally prove the security or vulnerability of DTSN using $crypt^{prob}_{time}$.

The attacker model $\mathcal{M}_A$: We assume that an attacker can intercept the outputs of the honest nodes on public channels, and modify them according to its knowledge and computation ability. The attacker's knowledge consists of the intercepted outputs during the protocol run and the information it can create, for instance, its private keys or fake data such as packet IDs, etc. The ability of the attacker(s) is that it can modify the elements of the plaintexts, such as the base number and the bits of the ACK/NACK messages, the EAR and RTX bits and sequence number in data packets. The attacker can also create entire data or control packets including data it posseses. Further, attacker(s) can send packets to its neighborhood. We also assume several attackers who can share information with each other.

To describe the activity of the attacker(s) we apply the concept of the environment, used in the applied π-calculus that model the presence of the attacker(s) in an implicit way. This concept enables us to model more than one attacker who can even cooperate with each other. Every message that is output on a public channel is available for the environment, that is, the environment can be seen as a group of attackers who can share information with each other, for instance, via a side channel. In particular, in our model we consider a specific topology of honest nodes.

We define the ideal version of the process *Prot(params)*, denoted by $Prot^{ideal}(params)$, which contains the ideal version of *DTSN(params)*:

/* The ideal version of the DTSN protocol for the given topology */
$Prot^{ideal}(params) \stackrel{def}{=}$
let $(e^s_1, e^s_2, e^s_3, e^i_1, e^i_2, e^i_3, e^d_1, e^d_2, e^d_3, cntsq) = (E, E, E, E, E, E, E, E, E, 1)$
in $INITDTSN^{ideal}()$;

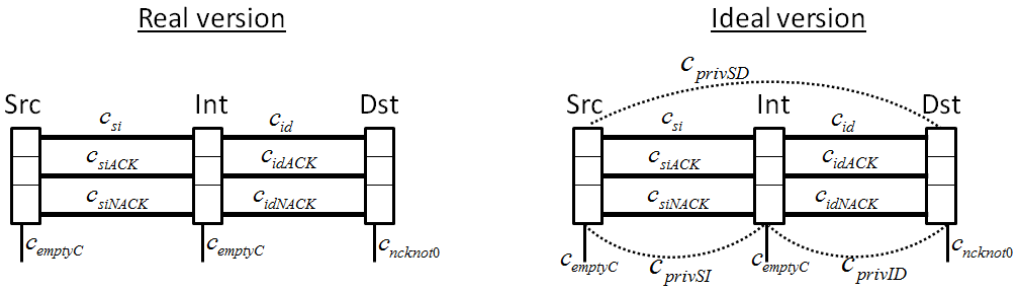

Figure 5: *The difference between the real and ideal version of DTSN and SDTP.*

where process $INITDTSN^{ideal}()$ contains $DTSN^{ideal}(params)$ instead of $DTSN(params)$. The main difference between $DTSN^{ideal}(params)$ and $DTSN(params)$ is that in $DTSN^{ideal}(params)$ honest nodes always are informed about what kind of packets or messages they should receive from the honest sender node. This can be achieved by defining hidden or private channels between honest parties, on which the communication cannot be observed by attacker(s). In Figure 5 we show the difference in more details. In the ideal case, three private channels are defined that are not available the attacker(s). *Src*, *Int* and *Dst* denote the source, intermediate and destination. Channels c_{privSD}, c_{privID} and c_{privSI} are defined between *Src* and *Dst*, *Int* and *Dst*, *Src* and *Int*, respectively. Whenever S sends a packet *pck* on public channel c_{si}, it also informs I about what should I receive, by sending at the same time *pck* directly via private channel c_{privSI} to I, so when I receives a packet via c_{si} it compares the message with *pck*. The same happens when I sends a packet to D. The channels c_{privSD} and c_{privID} can be used by the destination to inform S and I about the messages to be retransmitted. We recall that the communication via a private channel is not observable by the environment, hence, it can be seen as a silent τ transition. Note that for simplicity we omitted to include the upper layer and channel c_{sup} in the Figure. Finally, we also add additional public channels c_{emptyC} and $c_{ncknot0}$ for signalling that the cache has been emptied, and that the number of packets to be re-transmitted at the destination is larger than 0, respectively. These additional channels are defined for applying bisimilarities (observational equivalence) in the security proofs.

With this definition of $Prot^{ideal}(params)$ we ensure that the source and intermediate nodes are not susceptible to the modification or forging of ACK and NACK messages since they make the correct decision either on retransmitting or deleting the stored packets independently from the content of the control/data messages. Therefore, to show that DTSN is vulnerable to the modification or forging of ACK and NACK messages we prove that the running of $Prot(params)$ and $Prot^{ideal}(params)$ are not probabilistic timed bisimilar.

First of all, in $Prot(params)$ and $Prot^{ideal}(params)$ the source and intermediate nodes output the constants *CacheEmptyS* and *CacheEmptyI* respectively whenever they have emptied their buffers after processing an ACK or NACK message. This is defined by the following $crypt^{prob}_{time}$ code fragment (for $i \in \{1,2,3\}$):

$checkEi$(s, d, apID, e^s_{1-3}, sID, acknum) $\stackrel{def}{=}$
/* *Cache entry* e^s_i *is emptied and the number of the empty caches is increased* */
let $(e^s_i, nbrEcacheS) = (E, inc(nbrEcacheS))$ *in*
. . . . . .
/* *Here we add the resetting of the two timers on process Src* */
$\{x_c^{act} \leq T_{act}, x_c^{ear} \leq T_{ear}\} \triangleright (x_c^{act} \leq T_{act}, x_c^{ear} \leq T_{ear}) \hookrightarrow$ *let* $(earAtmp = 0)$ *in*
$[emptycacheS = 3]$
/* *If the cache has been emptied then CacheEmptyS is output* */
$\overline{c_{emptyC}}\langle CacheEmptyS\rangle.\ \| x_c^{act}, x_c^{ear} \| \{ x_c^{act} \leq T_{act} \} \triangleright$
$(x_c^{act} \leq T_{act}) \hookrightarrow Src$(s, d, apID, e^s_{1-3}, sID, earAtmp)
else $\| x_c^{act}, x_c^{ear} \| Src$(s, d, apID, e^s_{1-3}, sID, earAtmp);

In case of the source node (process *Src*), outputting of the constant *CacheEmptyS* is placed at the end of processes *checkE1*, *checkE2* and *checkE3*, and the processes that handles a NACK message. The constant *CacheEmptyS* is output whenever the number of the empty cache entries, *emptycacheS*, is 3, which means that the cache is emptied.

With the analogous concept, in process *Int* the output of the constant *CacheEmptyI* is placed at the end of the corresponding processes *checkE1I*, *checkE2I*, *checkE3I*, and the NACK handling processes.

For the destination node we specify the process *Dst* such that *nackNbr*, which is the number of the message to be re-transmitted (or the number of bits within the NACK message to be sent), is output via a public channel $c_{ncknot0}$ (i) whenever the source outputs the constant *CacheEmptyS*, and (ii) ($nackNbr > 0$) hold at the same time. This is solved by the following $crypt^{prob}_{time}$ code part in process *Dst*.

$$[nackNbr > 0]\ c_{emptyC}(= CacheEmptyS).\overline{c_{ncknot0}}\langle nackNbr\rangle.$$
$$Dst(e^{d}_{1-3}, \text{ackNbr, nackNbr, toRTX1, nxtsq});$$

This process says that if ($nackNbr > 0$) and *Dst* receives the constant *CacheEmptyS* on channel c_{emptyC} then it outputs *nackNbr* on channel $c_{ncknot0}$, followed by invoking recursively process *Dst*.

Let process *Prot'*(*params*) be a process such that its frame φ(*Prot'*(*params*)) contains the substitution $\sigma_1 = \{\ldots, CacheEmptyS/x_i, nackNbr/x_j, \ldots\}$ which captures the state that *CacheEmptyS* and *nackNbr* have been output right after each other; or $\sigma_2 = \{\ldots, CacheEmptyI/y_i, nackNbr/y_j, CacheEmptyS/x_i, nackNbr/x_j, \ldots\}$. In case of σ_1 the fact that *CacheEmptyS* and *nackNbr* is next to each other in this order, represents the state when all the cache entries of the source have been deleted and the number of packets to be retransmitted is greater than zero, and σ_2 means that both the caches of the intermediate and source node has been emptied, however, the number of packets to be retransmitted is greater than zero.

Both σ_1 and σ_2 represent an undesired situation because according to the specification of DTSN the source should store the packets to be retransmitted when a NACK packet is sent by the destination.

To capture the above mentioned situations as security holes, we need to modify the definition of static equivalence by adding the following problem specific requirements to the base definition of static equivalence (defined in Definition 1):

Definition 13. ***(Modified static equivalence for DTSN and SDTP)*** *Two extended processes A_1 and A_2 are statically equivalent, denoted as $A_1 \approx_s A_2$, if their frames are statically equivalent. Two frames φ_1 and φ_2 are statically equivalent if they includes the same number of active substitutions and same domain; and any two terms that are equal in φ_1 are equal in φ_2 as well.* ***In addition, we require that***

- *if $\sigma_1 = \{\ldots, CacheEmptyS/x_i, nackNbr/x_j, \ldots\}$ then σ_2 also includes the outputs of CacheEmptyS and nackNbr right after each other in this order.*
- *if $\sigma_1 = \{\ldots, CacheEmptyI/y_i, nackNbr/y_j, CacheEmptyS/x_i, nackNbr/x_j, \ldots\}$ then σ_2 also includes the outputs of CacheEmptyI, nackNbr and CacheEmptyS right after each other in this order.*

Beside this definition we have the following Lemma, which declares the insecurity of DTSN:

Lemma 1. *Besides the defined attacker model $\mathcal{M}_A$, the DTSN protocol is insecure against message manipulation attacks.*

According to Definition 9 processes *Prot*(*params*) and $Prot^{ideal}$(*params*) are not probabilistic timed bisimilar because each point of the definition is violated. According to the semantics of $crypt^{prob}_{time}$, let define states $s = (Prot(params), v)$ and $s^{ideal} = (Prot^{ideal}(params), v)$, where v denotes the initial clock valuation (i.e., when the two clocks x^{act}_c and x^{ear}_c are set to zero) at both states s and s^{ideal}. Further, let $s' = (Prot'(params), v')$, where *Prot'*(*params*) and v' is the process

that represents one of the above discussed undesired states, and the corresponding valuation, respectively. We assume that starting from s and s^{ideal} the Definition 9 is satisfied until some state pair (s_i, s_i^{ideal}) on the execution path from s and s^{ideal}, respectively. Finally, we assume that each operation (verification, sending, receiving on public channel) takes an equal amount of time d. We assume that sending and receiving on private/hidden channels c_{prvSI}, c_{prvDI}, and c_{prvSD} takes zero time amount. The first reason of this assumption is that, in the ideal version, we want the sending/receiving on each public channel and the corresponding private channel being performed at once. Specifically, the sending (receiving) on public channels and its private counterparts are placed next to each other, for example: $\overline{c_{privSI}}\langle t_{msg}\rangle.\overline{c_{si}}\langle t_{msg}\rangle$, and $c_{privSI}(x^{rcv}_{should}).c_{si}(x^{rcv}_{really})$. The second reason is that for the proofs, we want to assure that the transition $\overline{c_{si}}\langle t_{msg}\rangle$, in the real system, is timed simulated by $\overline{c_{privSI}}\langle t_{msg}\rangle.\overline{c_{si}}\langle t_{msg}\rangle$ in the ideal system. We note that the sending/receiving perfomed on the private channels is semantically described by a silent transition, that is, they cannot be observed by the environment.

In case of DTSN and SDTP protocols the set of distributions contains only one distribution π, $\Pi = \{\pi\}$, which is valid either in case of *Prot*(*params*) or $Prot^{ideal}$(*params*). Hence, in every case the scheduler F defined for DTSN and SDTP chooses π at each transition step. In order to prove the insecurity of DTSN, without lost of generality, we define π such that it chooses the transition leads to state (*strAndFwI*, v_k) with probability p, and to (*FwI*, v_k) with $1-p$. All the other transitions are chosen with probability 1, after resolving the choices, if any.
We define the function f that returns $\sum d_j$ itself, and we assume that every action takes an equal amount of time, say d. Due to the requirement $s' \mathcal{R} s'^{ideal}$, the processes in the two states have to be static equivalent (the first point of Definition 9), which means (according to the definition of static equivelance) that the frame of the proces in s'^{ideal} has to contain the constants *nackNbr*, *CacheEmptyS* and *CacheEmptyI*. Recall that to be static equivalence it is required that these constants are in the **same places as in the** σ_1 **or** σ_2 (Definition 13).

To explain the proof we denote the source, intermediate and destination node by S, I and D. We distinguish three types of attacks that concern (separately) the violation of the three points of Definition 9, respectively.

Scenario 1 (SC-1): In the first attack scenario the third point of Definition 9 is violated. The following execution from state s cannot be simulated by any execution trace from s^{ideal} in terms of probabilistic timed bisimilarity.

- $s \xrightarrow{\alpha_1(d),\ \pi}_{PTTS1} s_1$, where $\alpha_1 = \nu z_1.\overline{c_{sup}}\langle z_1\rangle$ with $\{1/z_1\}$. The upper layer requests the source to forward the packet with sequence number 1.

- $s_1 \xrightarrow{\alpha_2(d),\ \pi}_{PTTS1} s_2$, where $\alpha_2 = c_{sup}(z_1)$. The source S receives the request to forward the packet with sequence number 1.

- $s'_2 \xrightarrow{\alpha_3(d),\ \pi}_{PTTS1} s_3$, where $\alpha_3 = \nu z_2.\overline{c_{si}}\langle z_2\rangle$ with $\{(s,\ d,\ apID,\ sID,\ 1,\ 0,\ 0)/z_2\}$. State s'_2 can be reached from s_2 via silent transitions that model the verification steps. The source sends (s, d, *apID*, *sID*, 1, 0, 0) to the intermediate node. This message can be obtained by the attacker(s) (i.e., the environment).

- $s_3 \xrightarrow{\alpha_4(d),\ \pi}_{PTTS1} s_4$, where $\alpha_4 = c_{siACK}(5(z_2))$, and $5(z_2)$ represents the 5th element of the packet, which is the sequence number 1. The packet sent by S in the 2nd point is intercepted by the attacker(s) who, instead of forwarding it, sends an ACK with base number 1 to S. S received this message on c_{siACK} at state s_4.

- $s'_4 \xrightarrow{\alpha_5(d),\ \pi}_{PTTS1} s_5$, where $\alpha_5 = \nu z_3.\overline{c_{emptyC}}\langle z_3\rangle$ with $\{CacheEmptyS/z_3\}$. State s'_4 can be reached from s_4 via silent transitions. The constant *CacheEmptyS* is output by S after it has erased all of its cache entries.

Based on this action transition trace, we can see that $Prob^F(s'_4 \xrightarrow{\alpha_5(d),\ \pi}_{PTTS1} s_5) > 0$ while there is no any s_5^{ideal} such that $Prob^F(s_4^{ideal\ '} \stackrel{\alpha_5(d),\ \pi}{\Longrightarrow}_{PTTS2} s_5^{ideal}) > 0$, which violates the third point of Definition 9. We note that from s to s'_4 $Prot^{ideal}(params)$ can simulate $Prot(params)$ via the same actions and the corresponding states from s^{ideal} to $s_4^{ideal\ '}$.

Intuitively, in $Prot(params)$ the attacker can achieve that S or I (or both) empties its cache (via some action trace) with probability $pr > 0$, which cannot be simulated (via the corresponding action trace) in the ideal process $Prot^{ideal}(params)$, in which S or I (or both) empties its cache only with zero probability. This kind of attack scenario comes with the topologies $S - A - I - D$ or $S - I - A - D$, where A represents a compromised node. In the first topology, A does not forward the packets arrived from S, instead, it sends a fake ACK message with a large base number to S. This results in deleting the buffer of S. Similarly, in the second topology A can make I and S erase their buffers. It is critical that the attacker can be successful without the presence of the destination.

7.2 Security Analysis of the SDTP protocol

We define the ideal version of process $ProtSDTP(params)$, denoted by $ProtSDTP^{ideal}(params)$, in the same concept as in $Prot^{ideal}(params)$. The only difference is that in SDTP, the processes Src and Int are defined such that whenever the verification made by S and I on the received ACK/NACK message has failed, S and I output a predefined constant $BadControl$ via the public channel c_{badpck}. Note that this extension does not affect the correctness of SDTP, and only plays a role in the proofs of probabilistic timed bisimilarity.

Since the main purpose of SDTP is using cryptographic means to patch the security holes of DTSN, we examine the security of SDTP according to each discussed attack scenario to which DTSN is vulnerable.

Scenario 1 (SC-1): First, we prove that SDTP is not vulnerable to the attack scenario (SC-1) by showing that $ProtSDTP^{ideal}(params)$ can simulate (according to Definition 9) the transition trace produced by $ProtSDTP(params)$. Let us consider the action traces given in case of DTSN.

- The transition $s \xrightarrow{\alpha_1(d),\ \pi}_{PTTS1} s_1$, where $\alpha_1 = \nu z_1.\overline{c_{sup}}\langle z_1\rangle$ with $\{1/z_1\}$, can be simulated by the transition $s^{ideal} \xrightarrow{\alpha_1(d),\ \pi}_{PTTS2} s_1^{ideal}$ in $ProtSDTP^{ideal}(params)$.

- $s_1 \xrightarrow{\alpha_2(d),\ \pi}_{PTTS1} s_2$, where $\alpha_2 = c_{sup}(z_1)$, can be simulated by the corresponding transition $s_1^{ideal} \xrightarrow{\alpha_2(d),\ \pi}_{PTTS2} s_2^{ideal}$.

- $s'_2 \xrightarrow{\alpha_3(d),\ \pi}_{PTTS1} s_3$, where $\alpha_3 = \nu z_2.\overline{c_{si}}\langle z_2\rangle$ with $\{(s,\ d,\ apID,\ sID,\ 1,\ 0,\ 0,\ ACKMAC_1,\ NACKMAC_1)/z_2\}$, can be simulated by the two transitions $s_2^{ideal\ '} \xrightarrow{\tau(0),\ \pi}_{PTTS2} s_2^{ideal\ ''} \xrightarrow{\alpha_3(d),\ \pi}_{PTTS2} s_3^{ideal}$. The first transition is a silent transition and represents sending on the hidden channel c_{prvSI}. We note that in case of SDTP the packet sent by S includes the ACK MAC and NACK MAC. This packet is available for the attacker(s) who can manipulate it and send it to the neighbor nodes.

- In DTSN, the next transition in $ProtSDTP(params)$ is $s_3 \xrightarrow{\alpha_4(d),\ \pi}_{PTTS1} s_4$, where $\alpha_4 = c_{siACK}(t)$, describes that the attacker sends the ACK message to S with some content t. In DTSN t was $5(z_2)$, however, in SDTP the format of ACK also includes an ACK key. In general, t can be defined by $f_a(\mathcal{K} \cup z_2)$, where f_a is a subset of functions that define the operations performed by the attacker using its knowledge base $\mathcal{K} \cup z_2$ ($\mathcal{K}$ is its knowledge, which is extended constantly during the protocol run). It can be shown that for all possible behaviors ($f_a \subseteq \mathcal{B}$, where $\mathcal{B}$ describes attacker's computation ability, defined by the set

of functions available for the attacker), *ProtSDTP*^{ideal}*(params)* can simulate this with the corresponding transition $s_3^{ideal} \xrightarrow{\alpha_4(d),\ \pi}_{PTTS2} s_4^{ideal}$.

- Due to the fact that the ACK key of the packet sent by S has not been output yet on a public channel, the attacker cannot construct the correct ACK message for the packet. Formally, we can say that $K_{ack} \notin \mathcal{K} \cup z_2$ and $\mathcal{B}$ does not contain any function that returns the correct ACK/NACK keys for the packets sent by the source, hence, $f_a(\mathcal{K} \cup z_2)$ cannot be the correct ACK message for the first packet sent by S.

  Therefore, for any t in *ProtSDTP(params)* we have the transition $s_4' \xrightarrow{\alpha_5(d),\ \pi}_{PTTS1} s_5$, where $\alpha_5 = \nu z_3.\overline{c_{badpck}}\langle z_3\rangle$ with $\{BadControl/z_3\}$, which can be simulated by a transition $s_4^{ideal\ '} \xrightarrow{\alpha_5(d),\ \pi}_{PTTS2} s_5^{ideal}$ in *ProtSDTP*^{ideal}*(params)*.

Note that in SDTP, regarding the attack trace against DTSN, when S received the incorrect ACK sent by A, instead of deleting its buffer and outputing *CacheEmptyS*, the constant *BadControl* is output because of ACK MAC verification fail, in both the real and ideal systems.

Nevertheless, based on Definition 12, the SDTP is not secure because the real and ideal processes are not probabilistic timed bisimilar.

Lemma 2. *The SDTP protocol is insecure besides the attacker model* $\mathcal{M}_A$.

Since there can be the case when only the attacker sends a message, there is not any communication on private/hidden channels. To handle this situation, in each process of the ideal version, besides the receiving process of form $c_{privCH}(x^{rcv}_{should}).c_{ch}(x^{rcv}_{really}).Proc1$, we also define an another choice option $c_{ch}(x^{rcv}_{really}).Proc2$. In particular:

$$c_{prvCH}(x^{rcv}_{should}).c_{ch}(x^{rcv}_{really}).Proc1\ [\,]\ c_{ch}(x^{rcv}).Proc2$$

where c_{ch} and c_{prvCH} represent a public channel and its private counterpart. In *Proc1* the comparison of the two received messages is performed, and the node continues its operation according to the protocol in case they are the same, otherwise, it interrupts its normal operation. When the process $c_{ch}(x^{rcv}).Proc2$ is "activated" the node interrupts its normal operation, because it knows that the packet has not been sent by a honest node.

To prove the vulnernability of SDTP using probabilistic timed bisimilarity, we relax the definition of the ideal version such that the honest nodes only compare the received ACK/NACK messages with the expected ones. When they receive a data packet they proceed in the same way as the real version, namely, without any comparion with the expected message. Formally, in the above process, c_{prvCH} and c_{ch} range over the channels for ACK and NACK messages.

To prove Lemma 2, we show that the following action traces in the real version of the SDTP protocol cannot be simulated in the ideal version. The trace desribes the topology $S - A1 - I - A2$. For the sake of brevity, we omit to detail the transitions describing the first data packet sent from S to $A1$, and the request from uppler layer to S. The first transition that we consider below is when $A1$ has received the first packet from S.

1. $s \xrightarrow{\alpha_1(d),\ \pi}_{PTTS1} s_1$, where $\alpha_1 = c_{si}(t_{pck})$ with t_{pck} is (*s, d, apID, sID, 1, 0, 0,* $ACKMAC_{att}$, $NACKMAC_{att}$). $A1$ replaces the MACs of S by its computed MACs on the same data, and sends the modified packet to I. α_1 means that the packet is received by I.

2. The trace $s_1 \xrightarrow{\tau(d),\ 1}_{PTTS1} \ldots\ s_1^i \xrightarrow{\tau(d),\ p}_{PTTS1} s_1^{i+1} \ldots \xrightarrow{\tau(d),\ 1}_{PTTS1} s_1'$, is a series of silent transitions modelling internal computations made by I. $s_1^i \xrightarrow{\tau(d),\ p}_{PTTS1} s_1^{i+1}$ says that the received packet is stored (with probability p) by I. These can be simulated by the corresponding silent transitions.

3. Node I forwards the packet to $A2$ via the transition $s'_1 \xrightarrow{\alpha_2(d),\ \pi}_{PTTS1} s_2$, where $\alpha_2 = \nu z_1.\overline{c_{id}}\langle z_1\rangle$ with $\{(s,\ d,\ apID,\ sID,\ 1,\ 0,\ 0,\ ACKMAC_{att},\ NACKMAC_{att})/z_1\}$, which can be simulated by the transition $s_1^{ideal\ '} \xrightarrow{\alpha_2(d),\ \pi}_{PTTS2} s_2^{ideal}$ in $ProtSDTP^{ideal}(params)$.

4. $s_2 \xrightarrow{\alpha_3(d),\ \pi}_{PTTS1} s_3$, where $\alpha_3 = c_{idACK}(t_{ack})$ with t_{ack} is $(1,\ ACKKEY_{att})$. $A2$ sends the ACK message for the data packet to I, including the correct ACK Key for the MAC $ACKMAC_{att}$. Label α_3 means that the ACK message is received by I. This can be simulated by the transition $s_2^{ideal\ '} \xrightarrow{\alpha_3(d),\ \pi}_{PTTS2} s_3^{ideal}$.

5. As the result, node I deletes its buffer and outputs the constant *CacheEmptyI* on the public channel c_{emptyC}. This transition cannot be simulated by any corresponding transition trace in $ProtSDTP^{ideal}(params)$. This is because the ACK sent by A2 will be received on $c_{idACK}(x^{rcv})$, then, node I interrupts its operation. Hence, *CacheEmptyI* will never be output.

8 Automated security verification using the PAT process analysis toolkit

Related Methods: SPIN model-checker [18] and UPPAAL [2] are general purpose model-checking tools. CPAL-ES [16], and ProVerif [3] are automatic verification tools developed for verifying security protocols. The main drawback of them is that they lack sematics and syntax for defining the systems that include probabilistic and real-time behavior. Hence, they *cannot be used* to verify WSN transport protocols such as DTSN and SDTP. PRISM model-checker [13] supports probabilistic and real time systems but its limited specification language does not enable us to verify protocols/sytems that may perform complex computations.
Our Method: Our method is based on the PAT process analysis toolkit. PAT [8] is a self-contained framework to specify and automatically verify different properties of concurrent (i.e. supporting parallel compositions construct), real-time systems with probabilistic behavior. It provides user friendly graphical interface, featured model editor and animated simulator for debugging purposes. PAT implements various state-of-the-art model checking techniques for different properties such as deadlock-freeness, divergence-freeness, reachability, LTL properties with fairness assumptions, refinement checking and probabilistic model checking. To handle large state space, the framework also includes many well-known model-checking optimization methods such as partial order reduction, symmetry reduction, parallel model checking, etc. An another advantage of PAT is that it allows user to build customized model checkers easily.

Currently it contains eleven modules to deal with problems in different domains including real time and probabilistic systems. PAT has been used to model and verify a variety of systems, such as distributed algorithms, and real-world systems like multi-lift and pacemaker systems. However, PAT (so far) does not provide syntax and semantics for specifying cryptographic primitives and operations, such as digital signature, MAC, encryptions and decryptions, one-way hash function. Hence, we model cryptographic operations used by SDTP in an abstract, simplified way. Note that the simplication has been made in an intuitive way, and does not endanger the correctness of the protocol.

PAT is basically designed as a general purpose tool, not specifically for security protocols or any specific problem. It provides a CSP [11] (the well-known process algebra) like syntax, but it is more expressive than CSP because it also includes the language constructs for time and probabilistic issues. PAT also provides programming elements like comminucation channels, array of variables and channels, similarly as Promela [12] (Process Meta Language), the specification language used by the SPIN [12] model-checker. PAT handles time in a tricky way, namely, instead of modelling clocks and clock resets in an explicit manner, to make the automatic verification

be more effective it applies an implicit representation of time (clocks) by defining specific time expressions such as TIMEOUT, WAITUNTIL, INTERRUPT, etc.

Next, we briefly introduce the features provided by the main modules of PAT that we use to verify the security of DTSN and SDTP.

Communicating Sequential Programs (CSP#) Module. The CSP# module supports a rich modelling language named CSP# (a modified variant of CSP) that combines high-level modeling operators like (conditional or non-deterministic) choices, interrupt, (alphabetized) parallel composition, interleaving, hiding, asynchronous message passing channel.

The high-level operators are based on the classic process algebra Communicating Sequential Processes (CSP). Beside keeping the original CSP as a sub-language, CSP# offering a connection to the data states and executable data operations.

Global constant is defined using the syntax

```
#define constname val
```

where *constname* is the name of the constant and *val* is the value of the constant. Variables and array can be defined as follows

```
1. var varname = val; 2. var arrayname = [val_1,..., val_n]; 3. var arrayname[n]
```

In PAT variables can take integer values. The first point defines the variable with name *varname* with the initial value *val*; the second point defines the fix size array with n values, and the third point declares the array of size n, where each element is initialized to 0. To assign values to specific elements in an array, event prefix is used as follows:

$$P() = assignvalEV\{arrayname[i] = val\} \text{ -> } \textbf{Skip},$$

where the assignment of the ith element of the array *arrayname* is performed within the scope of the event *assignvalEV*.

In PAT, process may communicate through message passing on channels. Channels, and output/input actions on a channel can be declared using the next syntax:

```
1. (declaration of channel channame): channel channame size;
2. (output of the msg tuple (m1,m2,m3) on channame): channame!m1.m2.m3;
3. (input a msg (m1,m2,m3) on the channel channame): channame?x1.x2.x3;
```

channel is a keyword for declaring channels only, *channame* is the channel name and *size* is the channel buffer size. It is important that a channel with buffer size 0 sends/receives messages synchronously. A process is a relevant specification element in PAT that is defined as an equation

```
P(x1, x2, ..., xn) = ProcExp;
```

where *ProcExp* defines the behavior of process P. PAT defines special processes to make coding be more convenient: Process ***Stop*** is the deadlock process that does nothing; process ***Skip*** terminates immediately and then behaves exactly the same as ***Stop***.

Events are defined in PAT to make debugging be more straightforward and to make the returned attack traces be more readable. A simple event is a name for representing an observation. Given a process P, the syntax ev -> P describes a process which performs ev first and then behaves as P. An event ev can be a simple event or can be attached with assignments which update global variables as in the following example, $ev\{x = x + 1;\}$ -> ***Stop***; where x is a global variable.

A *sequential composition* of two processes P and Q is written as $P;Q$ in which P starts first and Q starts only when P has finished. A (general) choice is written as P [ ] Q, which states that either P or Q may execute. If P performs an event first, then P takes control. Otherwise, Q takes control. Interleaving represents two processes which run concurrently, and is denoted by P ||| Q. Parallel composition represents two processes with barrier synchronization is written as P || Q, where || denotes parallel composition. Not like interleaving, P and Q may perform

lock-step synchronization, i.e., P and Q simultaneously perform an event. For instance, if P is a -> c -> ***Stop*** and Q is c -> ***Stop***, because c is both in the alphabet of P and Q, it becomes a synchronization barrier.

Assertion: An assertion is a query about the system behaviors. PAT provides queries for deadlock-freeness, divergence-freeness, deterministic, nonterminating, reachabiliy, respectively as in the following syntax:

```
1. #assert P() deadlockfree;  /* asks if P() is deadlock-free or not. */

2. #assert P() divergencefree;  /* asks if P() is divergence-free or not. */

3. #assert P() deterministic; /*  asks if P() is deterministic or not. */

4. #assert P() nonterminating; /* asks if P() is nonterminating or not. */

5. #assert P() reaches cond; /* asks if P() can reach a state where cond
                                is satisfied. */
```

PAT's model checker performs Depth-First-Search or Breath-First-Search algorithm to repeatedly explore unvisited states until a deadlock state (i.e., a state with no further move).

Linear Temporal Logic (LTL): PAT supports the full set of LTL syntax. Given a process $P()$, the following assertion asks whether $P()$ satisfies the LTL formula.

```
#assert P() |= F;
```

where F is an LTL formula whose syntax is defined as the following rules,

```
F = e | prop | [] F | <> F | X F | F1 U F2 | F1 R F2
```

where e is an event, prop is a pre-defined proposition, [ ] reads as "always", <> reads as "eventually", X reads as "next", U reads as "until" and R reads as "Release". For example, the following assertion asks whether the $P()$ can always eventually reach the state where *goal* evaluates to true.

```
#assert P() |= []<> goal;
```

A goal (badstate, goodstate, etc.) is a boolean expression, for example, if we want to define the goal that the value of x is 5, we write the following

```
#define goal (x==5);
```

In PAT the mathematical operations and expressions can be specified in the C like style. PAT supports FDR's approach for checking whether an implementation satisfies a specification or not. Differ from LTL assertions, an assertion for refinement compares the whole behaviors of a given process with another process, e.g., whether there is a subset relationship. There are in total 3 different notions of refinement relationship, which can be written in the following syntax.

```
/* whether P() refines Q() in the trace semantics; */
  #assert P() refines Q()
/* whether P() refines Q() in the stable failures semantics; */
  #assert P() refines<F> Q()
/* whether P() refines Q() in the failures divergence semantics; */
  #assert P() refines<FD> Q()
```

Real-Time System (RTS) Module. The RTS module in PAT enables us to specify and analyse real-time systems and verify time concerned properties. To make the automatic verification be more efficient, unlike timed automata that define explicit clock variables and capturing real-time constraints by explicitly setting/reseting clock variables, PAT defines several timed behavioral patterns are used to capture high-level quantitative timing requirements *wait*, *timeout*, *deadline*, *waituntil*, *timed interrupt*, *within*.

1. ***Wait***: A wait process, denoted by *Wait[t]*, delays the system execution for a period of t time units then terminates. For instance, process *Wait[t]*;P delays the starting time of P by exactly t time units.

2. ***Timeout***: Process P *timeout[t]* Q passes control to process Q if no event has occurred in process P before t time units have elapsed.

3. ***Timed Interrupt***: Process P *interrupt[t]* Q behaves as P until t time units elapse and then switches to Q. For instance, process ($ev1$ -> $ev2$ -> ...) *interrupt[t]* Q may engage in event $ev1$, $ev2$ ... as long as t time units haven't elapsed. Once t time units have elapsed, then the process transforms to Q.

4. ***Deadline***: Process P *deadline[t]* is constrained to terminate within t time units.

5. ***Within***: The within operator forces the process to make an observable move within the given time frame. For example, P *within[t]* says the first visible event of P must be engaged within t time units.

Probability RTS (PRTS) Module. The PRTS module supports means for analysing probabilistic real-timed systems by extending RTS module with probabilistic choices and assertions.

The most important extension added by the PRTS modul is the probabilistic choice (defined with the keyword ***pcase***):

```
prtsP = pcase {
         [prob1] : prtsQ1
         [prob2] : prtsQ2
                ...
         [probn] : prtsQn
        };
```

where *prtsP*, *prtsQ1*,..., *prtsQn* are PRTS processes which can be a normal process, a timed process, a probabilistic process or a probabilistic timed process. *prtsP* can proceed as *prtsQ1*, *prtsQ2*, ..., *prtsQn* with probability *prob1*, *prob2*, ..., *probn*, respectively.

For user's convenience, PAT supports another format of representing probabilities by using weights instead of probs in the pcase construct. In particular, instead of *prob1*, ..., *probn* we can define *weight1*, ..., *weightn*, respectively, such that the probability that *prtsP* proceeds as *prtsQ1* is *weight1* / (*weight1* + *weight2* + ... + *weightn*).

Probabilistic Assertions: A probabilistic assertion is a query about the system probabilistic behaviors. PAT provides queries for deadlock-freeness with probability, reachabiliy with probability, Linear Temporal Logic (LTL) with probability, and refinement checking with probability, respectively as in the following syntax:

```
1. #assert prtsP() deadlockfree with pmin/ pmax/ prob;

2. #assert prtsP() reaches cond with prob/ pmin/ pmax;

3. #assert prtsP() |= F with prob/ pmin/ pmax;

4. #assert prtsP() refines Spec() with prob/ pmin/ pmax;

5. #assert Implementation() refines<T> TimedSpec();
```

The first assertion asks the (min/max/both) probability that *prtsP()* is deadlock-free or not; the second assertion asks the (min/max/both) probability that *prtsP()* can reach a state at which

some given condition *cond* is satisfied; the third point asks the (min/max/both) probability that *prtsP()* satisfies the LTL formula F. PAT also supports refinement checking in case of probabilistic processes. The last assertion ask the probability that the system behaves under the constraint of the specification (i.e., an ideal version of a process). PRTS module also supports timed refinement checking. The fifth assertion allows user to define the specification which has real-time features and check if the implementation could work under the constraint of timed specification.

8.1 Specifying the ACT and the EAR timers in the PAT analysis toolkit

In this subsection, we discuss the approach for specifying timers using the high-level timed patterns defined in PAT. Unlike timed automata, in PAT there is no possibility for defining explicit clock variables and capturing real-time constraints by explicitly setting/reseting clock variables. Hence, specifying systems and protocols that launch and reset clocks, and defining timers, is a challenging task in PAT. In order to achieve an equivalent specification of timers, we have to use the high-level timed patterns of PAT in tricky manner. In the following, we will discuss each timed behavioral pattern defined in PAT, and we reason about whether it can be used for modelling timers defined in DTSN and SDTP.

1. ***Wait***: A wait process *Wait[t]*;P delays the starting time of P by exactly t time units. This cannot be used for the timers in DTSN and SDTP because the source node does not perform a delay during its operation.

2. ***Timeout***: Process P *timeout[t]* Q passes control to process Q if no event has occurred in P before t time units have elapsed. Otherwise, if there is any event occured within t, the process P is continued without switching to Q. This timed pattern can be used for the timers in DTSN and SDTP as follows: *SrcWaitsforPck*() *timeout[t]* *SrcAfterTimeout*(), where *SrcWaitsforPck*() and *SrcAfterTimeout*() specify the behavior of the source waiting for a message (packet), or when a timeout happens without receiving any message. Intuitively, this process says that if the source receives a message (a data, an ACK, or a $NACK$) during waiting for an event, which is the first visible event in *SrcWaitsforPck*(), then the source will continue its operation with processing the received message. Otherwise, the source will handle the expiration of the ACT or the EAR timer.

3. ***Deadline***: Process P *deadline[t]* is constrained to terminate within t time units. In DTSN and SDTP the nodes are not required to be finish within any time constraint, instead, they only change their behavior upon timeout. Hence, this timed pattern cannot be used in our case.

4. ***Within***: The within operator forces the process to make an observable action (event) within the given time frame. For example, P *within[t]* says the first visible event of P must be engaged within t time units. For the same reason like in case of *deadline*, this is not suitable for modelling the timers in DTSN and SDTP.

5. ***Timed Interrupt***: Process P *interrupt[t]* Q behaves as P until t time units and then switches to Q. Intuitively, timed interrupt could be used to model the timers in DTSN and SDTP, for instance with *SourceBeforeTimeout*() *interrupt[t]* *SourceAfterTimeout*(), where after timeout, *SourceAfterTimeout*() is activated. However, the main problem with this process is that within *SourceBeforeTimeout*(), the timer(s) can be reset, and there is no possibility for reset the value of t in *interrupt[t]*. In other words, we cannot "jump" outside the scope of *interrupt[t]*, hence, the reset performed in *SourceBeforeTimeout*() can be overwritten by *interrupt[t]*.

In the following, we show in details how to apply the *timeout* pattern in PAT to model the EAR and ACT timers. According to the definition of the source in DTSN and SDTP (Figure 1), the source checks the expiration of the EAR and the ACT timers, only when it returned to the status *Wait for event*. This means that within a branch corresponding to the received message (a data,

an *ACK*, a *NACK*), timer expirations do not interrupt the behavior of the source immediately. Moreover, whenever the source steps into one branch, in case of DTSN, the timers will be reset within the branch, before returning to the status *waiting for event*. This can be modelled by the following process in PAT:

```
Src(x, y) =
  1. SrcWaitsforPck() timeout[x] SrcAfterActTimeout()
     <>
  2. SrcWaitsforPck() timeout[y] SrcAfterEarTimeout();
```

The behavior of the source after both the *ACT* and *EAR* timers have already been launched can be modelled as a non-deterministic choice of the two processes that represent the case when the *ACT* timer (point 1), or the *EAR* timer (point 2) expires earlier than the other, respectively. The process variables x and y capture the value of the *ACT* and *EAR* timers, respectively.

```
SrcWaitsforPck() = SrcRcvPck() [] SrcRcvAck() [] SrcRcvNack();
```

SrcWaitsforPck() specifies the source waiting for a packet (a data, an *ACK*, a *NACK* packet). If within x (*ACT* timer) or y (*EAR* timer) time units no packet has been received, the source handles the corresponding timeout.

```
SrcRcvPck() = chSIPck?seq.ear.rtx -> HandlePck(seq,ear,rtx);

SrcRcvAck() = chSIAck?ack -> HandleAck(ack);

SrcRcvNack() = chSINack?nack -> HandleNack(nack);
```

Within each branch represented by the processes *HandlePck*(seq,ear,rtx), *HandleAck*(ack), and *HandleNack*(nack), the timers will be (certainly) reset in case of DTSN (Figure 1).

```
HandleAck(ack) =
    FreeBuff(ack); RstEarAtmpt(); SndEar(); Src(Tact, Tear);

HandleNack(nack) =
    RtxPckts(nack); FreeBuff(nack); RstEarAtmpt(); SndEar(); Src(Tact, Tear);
```

In process *HandleAck*(ack), according to the Figure 1, the source frees its buffer based on the received *ACK*, then the *EAR* is reset to zero, and the *EAR* packet is sent. Finally, both the *EAR* and the *ACT* timers are reset before the source returns to wait. We specify the timer resets by invoking the process *Src*(*Tact*, *Tear*). The specification of *SrcRcvPck*() is little more complicated because the reset of the *ACT* timer is not performed at the end of the branch, at the moment when the source returns to wait. Hence, we have to take into account the time elapsed since the *ACT* timer is reset (after the source stored the received data packet), until the source returns to wait. We assume that each operation sequence consumes a certain amount of time, namely:

- the branch *NotOutBufFull* → *NotAWmultiple* → *SndDt* consumes *T1* time units;
- the branch *NotOutBufFull* → *AWmultiple* → *SndEARPigyBack* → *SetEARAtmpTo1* consumes *T2* time units;
- the branch *OutBufFull* → *SndEARPigyBack* → *SetEARAtmpTo1* consumes *T3* time units.

Therefore, at the end of the three branches, the process *Src*(x, y) is called but with different time values:

```
HandlePck(seq,ear,rtx) = UntilActReset(); Branch1(); Src(Tact-T1, Tear);

HandlePck(seq,ear,rtx) = UntilActReset(); Branch2(); Src(Tact-T2, Tear);

HandlePck(seq,ear,rtx) = UntilActReset(); Branch3(); Src(Tact-T3, Tear);
```

Finally, the processes *SrcAfterEarTimeout*() and *SrcAfterActTimeout*() which specify how the source handles the EAR and ACT timeouts, respectively, are defined as follows:

```
SrcAfterEarTimeout() = incEAR{earatmpt = earatmpt+1;} -> EarAtmptLargerMax();

EarAtmptLargerMax() =
  if (earatmpt > MAXEARATT) {chErr!ERROR -> chSEnd!SESSIONEND -> Stop;}
    else {chEar!EAR -> Src(Tact, Tear)};
```

Process *SrcEarTimeout*() defines the behavior of the source after the EAR timer expired. According to the definition of the source node, first, the value of the EAR attempts is increased and then the source examines if the updated value exceeds the upperbound of the EAR attempts (this examination is specified by process *EarAtmptLargerMax*()). The else branch of *EarAtmptLargerMax*() says that if the value of the EAR attempts is less or equal to the maximal value, the source sends an EAR packet and then resets the EAR and ACT timers.

```
SrcActTimeOut() =
    if (OutBufL == 0) {chSEnd!SESSIONEND -> Stop}
     else if (OutBufL > 0 && EarPending == 1) {Src()}
      else if (OutBufL > 0 && EarPending == 0){chEar!EAR -> Src(Tact, Tear)};
```

Process *SrcActTimeOut*() specifies the behavior of the source node after an ACT timeout. Based on the definition of the source, first, the source checks if there is any unconfirmed data packet. Namely, the length (i.e., the occupied entries) of the output buffer is zero or not. In case all of the data packets have been confirmed, the source sends the session-end signal, "commanding" all the participant nodes to finish the session, and then it stops. If there is at least one unconfirmed data packet, the source checks whether it has sent an EAR and waiting for the corresponding ACK or $NACK$ (i.e., whether there is an EAR pending). If yes, the source returns to wait for event, which is modelled by invoking *Src*(*Tact*, *Tear*). Otherwise, the source sends an EAR packet and resets the EAR and ACT timers.

The case of SDTP is a bit different from DTSN, because in SDTP after receiving an ACK or a $NACK$, the source node performs MAC verifications. In case all the MAC verifications are successful, the source continues its operation and resets the timers like in DTSN. However, if only one MAC verification fails, then the source returns to wait without resetting any timers. Hence, in SDTP we have to consider the time elapsed due to the MAC verifications until the failed one. For this purpose, we assume that one MAC verification consumes $Tmac$ time units.

In case of the verification of an ACK packet fails the source returns to wait with the updated $timeout[Tact - Tmac]$ and $timeout[Tear - Tmac]$, that is, invoking $Src(Tact - Tmac, Tear - Tmac)$. When a MAC verification failed at the n-th verification step, the source returns to wait with the updated $timeout[Tact - n \times Tmac]$ and $timeout[Tear - n \times Tmac]$, that is, calling $Src(Tact - n \times Tmac, Tear - n \times Tmac)$. The case of successful verifications is similar as in DTSN. Of course if $(Tear - n \times Tmac)$ or $(Tact - n \times Tmac)$ becomes negative, then we choose the zero value instead.

8.2 On verifying DTSN using the PAT process analysis toolkit

Following the concept in Section 7, we define public (symmetric) channels between each node pair. The channels *chSIPck*, *chDIPck* are for transfering data packet, while *chSIAck*, *chSINack*, *chDIAck*, *chDINack*, *chSIEar*, and *chDIEar* are for ACK, NACK, and EAR messages, respectively. In

addition, we add channels *chUpSPck* and *chUpDPck* between the upper layer and S, D, respectively. We also define the channel *chEndSession* between the source and the other entities, for indicating the end of a session.

We define different constants such as the *ERROR* for signalling errors according to DTSN, and the time values of timers, the size of buffers, the maximal value of packet for a session, and the constant *EARPCK* that represents a stand-alone EAR packet. We assume that the probability that a packet sent by a node has been lost and does not reach the addresee, denoted by the constant *PLOST*. The probability that an intermediate node stores a packet, denoted by the constant *PSTORE*. The value of activity timer is denoted by the constant *TACT*, while the value of the ear timer is named by *TEAR*. Finally, we defined the maximal number of EAR attempts with the constant *MAXEAR*. We give each of these constants a value, which are only (intuitively meaningful) examples for building and running the program code.

One of the biggest advantage of PAT compared with the other solutions is that it supports probabilistic and timed, CSP-like behavioral syntax and semantics, which are important in our case. The main drawback of PAT is that it is not optimized for verifying security protocols in presence of adversaries, hence, in the current form it does not support a convenient way for modelling attackers. In PAT the attacker(s) are not included by default, and the user has to define the attacker's behavior and its place in the network explicitly.

To analyse the security of DTSN we define the attacker process(es) based on the following scenarios. We examine different places of the attacker(s) in the network: *Top1.* $S - A - I - D$; *Top2.* $S - I - A - D$; *Top3.* $S - A1 - I - A2 - D$. We recall the assumption that the source and destination cannot be the attacker. For each scenario, we define additional symmetric channels *chASPck*, *chASAck*, *chASNack*, *chASEAR*, *chADPck*, *chADAck*, *chADNack*, *chADEAR*, between the attacker(s) and its(their) honest neighbors.

The attacker model $\mathcal{M}_A^{PAT}$: In order to reduce the state space during the verification (for preventing run out of memory), we limit the attacker's ability according to the messages exchanged in DTSN. More specifically, the attacker intercepts every message sent by its neighbors, and it can modify the content of the intercepted packet as follows:

- it can increase, decrease or replace the sequence number in data packets;
- it can set/unset the EAR bit and RTX bit in each data packet;
- it can increase, decrease or replace the base (ack) number in ACK/NACK packets;
- it can change the bits in NACK packets;
- it can include or replace the correct MACs with the self computed MACs (with owned keys).
- the combination of these actions.

and finally it can forward the modified packets to the neighbor nodes. In addition, we we assume that an attacker has no memory, namely, it can construct messages only based on the latest information it receives, or its generated data. To check the security property based on bisimilarity and equivalence, we define the process *DTSNideal()* for the ideal version of DTSN, in a similar concept as in Section 7. We specify the following *bad states*, in the form of assertions and goals in PAT, which represent the insecurity of the protocol, and we run automatic verification to see whether these bad states can be reached.

Let us consider the topologies *Top1* and *Top2*. The first main design goal of DTSN is to provide reliable delivery of packets. Hence, if the attacker(s) $\mathcal{M}_A^{PAT}$ can achieve that the probability of delivery of some packet in a session (i.e., the probability of the delivery of all packets in a session) is zero, then we say that DTSN is not secure in the presence of the defined adversaries. The assertion, denoted by *violategoal1*, for verifying the security of DTSN regarding this first main goal is the following:

$$\textbf{\#define}\ violategoal1\ (OutBufL == 0\ \&\&\ BufI == 0\ \&\&\ numNACK > 0)$$

where the (global) variables *OutBufL*, *BufI* are the number of the occupied entries at the soure and intermediate node, respectively, while the variable *numNACK* is the number of the packet that the destination has not received, and are requiring to be retransmitted. Hence, (*OutBufL* == 0) and (*BufI* == 0) represent that the cache of S and I are emptied, but at the same time (*numNACK* > 0), which means that D has not received all of the packets.

A1. ***#assert*** *DTSN*() ***reaches*** *violategoal1*;
A2. ***#assert*** *DTSNideal*() ***reaches*** *violategoal1*;
A3. ***#assert*** *DTSN*() ***reaches*** *violategoal1 with pmax*;
A4. ***#assert*** *DTSNideal*() ***reaches*** *violategoal1 with pmax*;
A5. *DTSNidealHide*() ***refines*** *DTSNHide*().

For the topology *Top2*, we run the PAT model-checker with the default settings for (A1) and (A2) and get the following results: (A1) is ***Valid*** and (A2) is ***Not Valid***, which means that DTSN can be attacked, while the ideal version is not susceptible to this weakness. PAT also returns an attack scenario for A1. Running PAT for (A3) and (A4) we receive the result that the maximum probability of reaching *violategoal1* in *DTSN*() is greater than 0, while in case of *DTSNideal*() it is 0. Because in PAT the refinement check between two processes is based on the equivalence of their traces of visible events, hence, to make the verification of refinement defined in (A5) be meaningful, in *DTSNideal*() and *DTSN*() we have to hide (i.e. make invisible) the events that is not defined in the code of the another version. Note that we specify *DTSNideal*() and *DTSN*() in such a way that, making these events invisible is meaningful and the correctness of the verification is not corrupted.

Now let us consider the topology *Top3* that includes two attackers $A1$ and $A2$. we specify the bad states for DTSN and SDTP, and we run the model-checking to see if these bad states can be reached. The bad states, and the verification goals can be defined in PAT's language in the form of logical formulae and assertions, respectively.

Let the number of buffer entries that are freed at node I after receiving an ACK/NACK message, be *freenum*, and the number of packets received in sequence by node D, be *acknum*. The bad state *violategoal2* specifies the state where (*freenum* > *acknum*).

```
  PAT code:
    #define violategoal2 (freenum > acknum)
A6. #assert DTSNsubA1subA2() reaches violategoal2
A7. #assert DTSNA1A2() reaches violategoal2
```

In case the process *DTSNA1A2*() reaches *violategoal2*, it can be seen as a security hole or an undesired property of DTSN, because according to the definition of I, it should always empty at most as many cache entries as the ack number it receives in ACK/NACK messages (*acknum*).

Unfortunately, for one process, PAT only returns one attack trace, and always the same one. Hence, to obtain different attack scenarios we have to modify *DTSNA1A2*() by changing the ability of the attackers. In particular, besides *DTSNA1A2*(), which contains the full ability of the attackers, we limit their abilities by including different subprocceses of *procA1* and *procA2*, namely, *subA1* and *subA2*. We denote *DTSNsubA1subA2*() as the process which includes different sub-processes of *procA1*() and *procA2*(). Then, we examine if there is any *DTSNsubA1subA2*() that reaches *violategoal2*.

First, we define the attacker $A1$ such that 1.) it sends a data packet to I without receiving any message (this is defined by process *A1NotRcvSndPck2I*()); or after receiving a data packet *seq.ear.rtx* from S, 2.) it sends a data packet to I (defined by process *A1RcvPckSndPck2I*()); or 3.) it forwards the packet unchanged to I. The second attacker $A2$ is defined such that 1.) it sends an ACK to I without receiving any message (defined by process *A2NotRcvSndAck2I*()); or 2.) after receiving a data packet from I, it sends an ACK to I (defined by process *A2RcvPckSndAck2I*());

```
subA1() =
```

```
  A1NotRcvSndPck2I()
  [] chSAPck?seq.ear.rtx ->
     (
       A1RcvPckSndPck2I() [] chIAPck!seq.ear.rtx -> subA1()
     )

subA2() =
  A2NotRcvSndAck2I()
  [] chIAPck?seq.ear.rtx -> A2RcvPckSndAck2I()

DTSNsubA1subA2() =
  UpLayer() ||| procS() ||| subA1() ||| procI() ||| subA2() ||| procD()
```

After running the PAT model-checker for the assertion (B6), the tool returned *Valid* along with the following scenario:

1. A1 gets the first data from S, and changes the seq number 1 to seqA2
2. A1 sends I the packet seqA2.ear.rtx (chIAPck!seqA2.ear.rtx)
3. I stores this packet and forwards it to A2 (chIAPck!seqA2.ear.rtx)
4. A2, after obtaining this packet (chIAPck?seqA2.ear.rtx), sends to I the ACK with ack number seqA2 (chIAAck!seqA2)
5. I erases its entire buffer, while D has not received any data yet.

As the result, basically, the attackers $A1$ and $A2$ can always achieve that the buffer of I is emptied, because by definition, *seqA2* is the largest possible sequence number, hence will be larger than every seq number stored by I. In the worst case, node I is always prevented from caching packets which corrupts the design goal of DTSN. We note that for the assertion (A7), due to the complexity of the attackers in *DTSNA1A2*(), the model-checker returns *Valid* after a larger amount of time and with a much longer attack trace. This long trace essentially shows the same type of attack as the shorter one, detected in the first assertion.

Moreover, in DTSN, when we extend the subprocess *subA1()* above such that $A1$ also forwards the ACK message it received from I to node S PAT returns an attack scenario where both S and I empty their buffer.

8.3 On verifying SDTP using the PAT process analysis toolkit

Next we examine the security of SDTP using the PAT toolkit. First of all we give some important code parts in PAT's specification language that specifies the behavior of the SDTP protocol assuming the topologies *Top1*: $S - A_1 - I - D$, *Top2*: $S - I - A_2 - D$, and *Top3*: $S - A_1 - I - A_2 - D$.

As already mentioned earlier, PAT does not support language elements for specifying cryptographic primitives and operations in an explicit way. We specify the operation of SDTP with the implicit representation of MACs and ACK/NACK keys.

First, recall that in SDTP the per-packet ACK and NACK keys are generated as

$$K_{ACK}^{(n)} = \text{PRF}(K_{ACK};\ \text{“per packet ACK key”};\ n)$$
$$K_{ACK}^{(n)} = \text{PRF}(K_{KACK};\ \text{“per packet NACK key”};\ n).$$

Following this concept, in PAT we define the ACK key and NACK key for the packet with sequence number n by the “pair” ***n.Kack*** and ***n.Knack***, respectively. To reduce the verification complexity we made abstraction on the key generation procedure, and model the session

ACK/NACK master keys by the unique constants *Kack* and *Knack*. Then we specify the packets sent by the source node as follows: ***sq.ear.rtx.sq.sq.Kack.sq.sq.Knack***, where the first part *sq.ear.rtx* contains the packet's sequence number, the EAR and RTX bits, respectively; the second part *sq.sq.Kack* and the third part *sq.sq.Knack* represent the ACK MAC and NACK MAC computed over the packet *sq* without the EAR and RTX bits, using the per-packet ACK and NACK keys *sq.Kack*, and *sq.Knack*. An ACK message has the following forms: *acknbr.acknbr.Kack*, where *acknbr.Kack* is the correspoding ACK key of *acknbr*. A NACK message has the format *acknbr.nckb1.acknbr.Kack.nckb.Knack*, where *nckb.Knack* is the NACK key of the packet to be retransmitted, *nckb*. The NACK message can include more bits, in a similar concept.

By default, the attackers do not posses the two master keys *Kack* and *Knack* of honest nodes, but only their keys *Katt*. Because honest nodes are specified to waiting for these MACs format, the attackers should compose the MACs in this format as well, namely, *sqA.sqA.Katt*. The attackers cannot use the master keys to construct the per-packet ACK/NACK keys, and when it obtains a MAC, e.g., *sq.sq.Kack*, it cannot use *sq.Kack*, only in case it receives *sq.Kack* itself.

Now we turn to discuss the automatic verification of SDTP in PAT. First, to see if SDTP reaches its design goals in a hostile environment we define the ideal version of SDTP, in the same concept as in the manual analysis, in Section 7, and modelled it by process *SDTPideal()*.

Basically, the design goal of SDTP is to make DTSN achieve its goals in a hostile environment. Hence, we examine every assertion that has already been defined in case of DTSN. We specify the behavior of the attacker(s) in case of the DTSN protocol. We distinguish the following scenarios and examine the possible ability of the attacker(s). The behaviors of the attackers are defined as the processes *procA1()* and *procA2()*. In our model, by default the attackers have two sequence number *seqA1* and *seqA2* which are the smallest (i.e., 1) and the largest possible sequence numbers, respectively. The attackers can include $earA \in \{0, 1\}$, $rtxA \in \{0, 1\}$ in their data packets. The attackers, in addition, posses the pre-defined values *bA1*,..., *bA4* for requiring re-transmission in NACK messages.

Process *procA1()*, which defines the behavior of the first attacker $A1$, is specified as a external choice of the following four activities. Each of them is composed of additional choice options.

1. *Without receiving any message*: (i.) $A1$ sends a data packet, with *seqA1.earA.rtxA* or *seqA2.earA.rtxA*, to I; (ii.) $A1$ sends an ACK, for the packet *seqA1* or *seqA2*, to I or to S; (iii.) $A1$ sends a NACK, with the ack number *seqA1* or *seqA2*, and a combination of *bA1*, ..., *bA4*, to I or to S.

2. *After receiving a data packet* (*chSAPck?seq.ear.rtx*): (i.) $A1$ sends a data packet, with the sequence number *seq*, *seqA1* or *seqA2*, and different values ear/rtx bits, to I; (ii.) $A1$ sends an ACK, with the ack number *seq*, *seqA1* or *seqA2*, to I or S; (iii.) $A1$ sends a NACK, with the ack number *seq*, *seqA1* or *seqA2*, and a combination of *bA1*,..., *bA4*, to I or to S .

3. *After receiving an ACK* (*chIAAck?ack*): (i.) $A1$ sends a data packet, with the sequence number *ack*, *seqA1* or *seqA2*, to I; (ii.) $A1$ sends an ACK, with the ack number *ack*, *seqA1* or *seqA2*, to I or to S; (iii.) $A1$ sends a NACK, with the ack number *ack*, *seqA1* or *seqA2*, and a combination of *bA1*,..., *bA4*, to I or to S.

4. *After receiving a NACK with 1-4 bits* (*chIANack?ack.b1* [ ] *chIANack?ack.b1.b2* [ ] *chIANack?ack.b1.b2.b3* [ ] *chIANack?ack.b1.b2.b3.b4*): (i.) $A1$ sends a data packet, with with the sequence number *ack*, *seqA1* or *seqA2*, to I; (ii.) $A1$ sends an ACK, with the sequence number *ack*, *seqA1* or *seqA2*, and a combination of *bA1*,..., *bA4*, *b1*, *b2*, *b3*, *b4*, to I or to S; (iii.) $A1$ sends a NACK to I or to S. We recall that the attacker, besides the self-generated data, can only use the information in the latest received messages. Hence, when the attacker receives the NACK *ack.b1*, it can only use (besides its own data) *ack* and *b1*.

The DTSN protocol with the second topology is specified as the parallel compositions of each honest node and the attacker A1:

$$\text{DTSNA1()} = \text{procS()} \;|||\; \text{procA1()} \;|||\; \text{procI()} \;|||\; \text{procD()}$$

For the second topology S - I - A2 - D, the scenarios and PAT codes are the same as in the case of the first topology S - A1 - I - D except that the used channels at each corresponding step are changed as follows: In the first topology, A1 receives data packets from S on *chSAPck*, which is changed to *chIAPck* in the second case because now data packets come from I. Similarly, the inputs on *chSAAck* and *chSANack* are changed to *chDAAack* and *chDANack*, respectively. The outputs by A on *chIAPck*, *chIAAck*, *chIANack*, *chSAAck* and *chSANack* are changed to *chDAPck*, *chDAAck*, *chDANack*, *chIAAck* and *chIANack*, respectively. The attacker process *procA2()* describing the behavior of A2 is specified in the same way as *procA1()*, but with different channels.

The DTSN protocol with the second topology is specified as the following parallel compositions:

$$\text{DTSNA2()} = \text{UpLayer()} \;|||\; \text{procS()} \;|||\; \text{procI()} \;|||\; \text{procA2()} \;|||\; \text{procD()}$$

For the third topology S - A1 - I - A2 - D, we apply both the specification of processes A1 and A2. The DTSN protocol with the third topology is specified as the parallel compositions of each honest node and the two attackers:

$$\text{DTSNA1A2()} = \text{UpLayer()} \;|||\; \text{procS()} \;|||\; \text{procA1()} \;|||\; \text{procI()} \;|||\; \text{procA2()} \;|||\; \text{procD()}$$

For the SDTP protocol, there are the same three topologies and the attacker's behavior scenarios are specified similarly as in case of DTSN, however, with different message formats. We simply extend the process specification given in DTSN() with the corresponding ACK/NACK MACs and per-packet keys.

The following defined PAT codes is applied for asking if these bad states can be reached in SDTP:

B1. ***#assert*** *SDTP()* ***reaches*** *violategoal1p1*;
B2. ***#assert*** *SDTPideal()* ***reaches*** *violategoal1p1*;
B3. ***#assert*** *SDTP()* ***reaches*** *violategoal1p1 with pmax*;
B4. ***#assert*** *SDTPideal()* ***reaches*** *violategoal1p1 with pmax*;
B5. *SDTPidealHide()* ***refines*** *SDTPHide()*.

We run PAT to model-check (B1), which is related to the first main design goal of DTSN, for SDTP, and get the result ***Not Valid***. This means that in the presence of the same attacker(s), DTSN can be corrupted such that D has not received some packets and required retransmissions but the buffers of S and I are emptied, however, it cannot be happen in the SDTP protocol. Assertion (B2) also results in ***Not valid***. Assertions (B3), (B4) return 0 as the maximal probability *pmax*. Finally, checking (B5) also ends with *Valid*. We note that *SDTPidealHide*() and *SDTPHide*() are defined in the similar concepts as in DTSN case, hiding the events belonging to the communication on private channels.

In the following, we examine the vulnerability describes by the assertion *violategoal2* for SDTP. We take the same *violategoal2* and consider the similar assertions.

```
B6. #assert SDTPA1A2() reaches violategoal2
B7. #assert SDTPsubA1subA2() reaches violategoal2
```

SDTPA1A2() and *SDTPsubA1subA2()* are defined in the similar concept as in DTSN. We run the PAT model-checker with the attackers processes:

```
subA1() =
 /* A1 sends a data packet to I, without receiving any message, OR */
 A1NotRcvSndPck2I()
 /* After receiving a data packet on channel chSAPck */
 [] chSAPck?seq.ear.rtx.seq1.seq2.Kack.seq3.seq4.Knack ->
   (
  /* A1 sends a data packet to I, OR */
```

```
    A1RcvPckSndPck2I()
  /* A1 forwards the packet unchanged to I */
    [] chIAPck!seq.ear.rtx.seq1.seq2.Kack.seq3.seq4.Knack -> subA1()
   )

subA2() =
 /* A2 sends a data packet to I, without receiving any message, OR */
 A2NotRcvSndAck2I()
 /* After getting a data on channel chIAPck, A2 sends ACK with seqA2 to I */
 [] chIAPck?seq.ear.rtx.seq1.seq2.Kack.seq3.seq4.Knack -> A2RcvPckSndAck2I()


SDTPsubA1subA2() =
  UpLayer() ||| procS() ||| subA1() ||| procI() ||| subA2() ||| procD()
```

And like in DTSN, the tool returned *Valid* for the assertion (B6) along with the following trace:

1. A1 sends to I a data pck seqA2.ear.rtx with the corresponding ACK MACs:
   seqA2.seqA2.Katt and NACK MACs: seqA2.seqA2.Katt
2. I stores this pck and forwards it (unchanged) to A2.
3. A2 received this packet and sends to I the ACK for seqA2: seqA2.seqA2.Katt.seqA2.Katt with the 2 keys seqA2.Katt and seqA2.Katt
4. As result, I deletes all the packets stored in its buffer because the key seqA2.Katt and the MAC seqA2.seqA2.Katt match.

In summary, we get the result that both DTSN and SDTP are susceptible for this sandwich style attack scenario. The main reason for this weakness is that in SDTP the intermediate nodes do not verify the origin of the received messages, they only check if the stored ACK/NACK MACs match the received ACK/NACK keys.

Note that we also performed verification and showed other weaknesses of SDTP, the reader can find them in the Appendix.

9 Conclusion

In this paper, we addressed the problem of formal and automated security verification of WSN transport protocols that may perform cryptographic operations. The verification of this class of protocols is difficult because they typically consist of complex behavioral characteristics, such as real-time, probabilistic, and cryptographic operations. To solve this problem, we proposed a probabilistic timed calculus for cryptographic protocols, and demonstrated how to use this formal language for proving security or vulnerability of protocols. To the best of our knowledge, this is the first such process calculus that supports an expressive syntax and semantics, real-time, probabilistic, and cryptographic issues at the same time. Hence, it can be used to verify systems that involve these three properties. In addition, we proposed an automatic verification method, based on the PAT process analysis toolkit for this class of protocols. For demonstration purposes, we apply the proposed manual and automatic proof methods for verifying the security of DTSN and SDTP, which are two of the recently proposed WSN tranport protocols, and showed that (i.) DTSN is vulnerable to packet manipulation attacks, (ii.) SDTP successfully patched some security holes found in DTSN, and (iii.) SDTP is still vulnerable to the sandwitch type attack.

In the future, we focus on improving the automatic security verification for this class of systems/protocols. Currently we found that PAT is the most suitable tool because it enables us to

define concurrent, non-deterministic, real time, and probabilistic behavior of systems in a convenient way. However, in its current form it does not support (or only in a very limited way) cryptographic primitives and operations, as well as the behavior of strong (external or insider) attackers. Finally, we believe that our proposed methods can be applied for verifying other similar systems, which we will show in our follow up work.

10 Acknowledgement

The work described in this paper has been supported by the grant TAMOP - 4.2.2.B-10/12010-0009 at the Budapest University of Technology and Economics. The authors are very thankful to Dr. Levente Buttyán, for his encouragement, guidance, and supports.

References

[1] M. Abadi and A. Gordon, *A calculus for cryptographic protocols: the Spi calculus*, Tech. Report SRC RR 149, Digital Equipment Corporation, Systems Research Center, January 1998.

[2] J. Bengtsson and F. Larsson, *Uppaal a tool for automatic verification of real-time systems*, Technical Report,Uppsala University, (96/67) (1996).

[3] Bruno Blanchet, *Automatic Proof of Strong Secrecy for Security Protocols*, IEEE Symposium on Security and Privacy (Oakland, California), May 2004, pp. 86–100.

[4] L. Buttyan and L. Csik, *Security analysis of reliable transport layer protocols for wireless sensor networks*, Proceedings of the IEEE Workshop on Sensor Networks and Systems for Pervasive Computing (PerSeNS) (Mannheim, Germany), March 2010, pp. 1–6.

[5] L. Buttyan and A. M. Grilo, *A Secure Distributed Transport Protocol for Wireless Sensor Networks*, IEEE International Conference on Communications (Kyoto, Japan), June 2011, pp. 1–6.

[6] Pedro R. D'Argenio and Ed Brinksma, *A calculus for timed automata*, Tech. report, Theoretical Computer Science, 1996.

[7] T. Dierks and E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.2*, RFC 5246, Internet Engineering Task Force, August 2008.

[8] Liu Yang et. al., *Pat: process analysis toolkit.*

[9] C. Fournet and M. Abadi, *Mobile values, new names, and secure communication*, In Proceedings of the 28th ACM Symposium on Principles of Programming, POPL'01, 2001, pp. 104–115.

[10] Jean Goubault-larrecq, Catuscia Palamidessi, and Angelo Troina, *A probabilistic applied pi-calculus*, 2007.

[11] C. A. R. Hoare, *Communicating sequential processes*, Commun. ACM **21** (1978), no. 8, 666–677.

[12] Gerard Holzmann, *Spin model checker, the: primer and reference manual*, first ed., Addison-Wesley Professional, 2003.

[13] M. Kwiatkowska, G. Norman, and D. Parker, *PRISM 4.0: Verification of probabilistic real-time systems*, Proc. 23rd International Conference on Computer Aided Verification (CAV'11) (G. Gopalakrishnan and S. Qadeer, eds.), LNCS, vol. 6806, Springer, 2011, pp. 585–591.

[14] Ruggero Lanotte, Andrea Maggiolo-Schettini, and Angelo Troina, *Weak bisimulation for probabilistic timed automata*, PROC. OF SEFM03, IEEE CS, Press, 2003, pp. 34–43.

[15] B. Marchi, A. Grilo, and M. Nunes, *DTSN - distributed transport for sensor networks*, Proceedings of the IEEE Symposium on Computers and Communications (Aveiro, Portugal), July 2007, pp. 165–172.

[16] John D. Marshall, II, and Xin Yuan, *An analysis of the secure routing protocol for mobile ad hoc network route discovery: Using intuitive reasoning and formal verification to identify flaws*, Tech. report, THE FLORIDA STATE UNIVERSITY, 2003.

[17] R. Milner, J. Parrow, and D. Walker, *A calculus of mobile processes, parts i and ii*, Information and Computation (1992).

[18] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and J.D. Tygar, *SPINS: security protocols for sensor networks*, ACM MobiCom (Rome, Italy), July 2001.

[19] F. Rocha, A. Grilo, P. Rogrio Pereira, M. Serafim Nunes, and A. Casaca, *Performance evaluation of DTSN in wireless sensor networks*, EuroNGI - Network of Excellence Workshop (Barcelona, Spain), Jan. 2008, pp. 1–9.

[20] J. Yicka, B. Mukherjeea, and D. Ghosal, *Wireless sensor network survey*, Computer Networks **52** (2008), no. 12, 2292–2330.

A The detailed specification of DTSN in $\boldsymbol{crypt}^{prob}_{time}$

In this section, we provide the detailed description (of the real version) of the DTSN protocol.

$Prot(params) \stackrel{def}{=}$
$\quad let\ (e^s_1, e^s_2, e^s_3, e^i_1, e^i_2, e^i_3, e^d_1, e^d_2, e^d_3, cntsq) = (E, E, E, E, E, E, E, E, E, 1)$
$\quad in\ INITDTSN()$;

$INITDTSN() \stackrel{def}{=} \overline{c_{sup}}\langle cntsq\rangle.DTSN(params)$

$DTSN(params) \stackrel{def}{=}$
$\quad upLayer(incr(cntsq)) \mid initSrc(\text{s, d, apID, } e^s_{1-3}, \text{ sID, earAtmp}) \mid$
$\quad Int(e^i_{1-3}) \mid Dst(e^d_{1-3}, \text{ ackNbr, nackNbr, toRTX1, nxtsq})$;

==========

Process that models the behavior of the upper layer (application that uses DTSN):

$upLayer(\text{cntsq}) \stackrel{def}{=}$
$\quad c_{sup}(= PRVOK).hndlePck(\text{cntsq})\ [\ ]\ c_{error}(= ERROR).upLayer(decr(cntsq))$
$\quad [\ ]\ c_{sessionEND}(= SEND).\mathbf{nil}\ [\ ]\ c_{dup}(x_{pck}).upLayer(cntsq)$;

$hndlePck(\text{cntsq}) \stackrel{def}{=}$
$\quad [cntsq \leq mxSQ]\ (\overline{c_{sup}}\langle cntsq\rangle.upLayer(incr(cntsq)))\ else\ upLayer(\text{cntsq})$;

The source handling the activity timer expiration:

$InitSrc(\text{s, d, apID, } e^s_{1-3}, \text{ sID, earAtmp}) \stackrel{def}{=}$
$c_{sup}(x_{sq}).($
$\quad \{\ x^{act}_c \leq T_{act}\ \} \triangleright (x^{act}_c \leq T_{act}) \hookrightarrow initFwdDt(\text{s, d, apID, } e^s_{1-3}, \text{ sID, } x_{sq})$
$\quad [\ ]\ \{x^{act}_c \leq T_{act}\} \triangleright (x^{act}_c \leq T_{act}) \hookrightarrow initRcvACKS(\text{s, d, apID, } e^s_{1-3}, \text{ sID, earAtmp})$
$\quad [\ ]\ \{x^{act}_c \leq T_{act}\} \triangleright (x^{act}_c \leq T_{act}) \hookrightarrow initRcvNACKS(\text{s, d, apID, } e^s_{1-3}, \text{ sID, earAtmp})$
$)$

$[\,]\ \{x_c^{act} \leq T_{act}\} \triangleright (x_c^{act} \leq T_{act}) \hookrightarrow c_{sessionEND}(= SEND).\mathbf{nil}$
$[\,]\ (x_c^{act} \geq T_{act}) \hookrightarrow actTimeOut;$

The source stores the received packet and resets the ACT timer:

$initFwdDt(\text{s, d, apID, } e^s_{1-3}\text{, sID, sq}) \stackrel{def}{=}$
. . . . . .
$let\ e_i^s = (s, d, apID, sID, sq)\ in\ \|\ x_c^{act}\ \|\ \{\ x_c^{act} \leq T_{act}\ \} \triangleright (x_c^{act} \leq T_{act})$
$\hookrightarrow checkBuffandAW(\text{s, d, apID, } e^s_{1-3}\text{, sID, sq, earAtmp})$

$checkBuffandAW(\text{s, d, apID, } e^s_{1-3}\text{, sID, sq, earAtmp}) \stackrel{def}{=}$
$[sq = AW]\ (\ let\ (ear,\ rtx,\ earAtmp) = (1,\ 0,\ 1)\ in\ \overline{c_{si}}\langle s, d, apID, sID, sq, ear, rtx\rangle.$
$Src(\text{s, d, apID, } e^s_{1-3}\text{, sID})\ else$
$(let\ (ear,\ rtx) = (0,0)\ in\ \overline{c_{si}}\langle s, d, apID, sID, sq, ear, rtx\rangle.\ Src(\text{s, d, apID, } e^s_{1-3}\text{, sID, earAtmp})$
$[\,]\ c_{sessionEND}(= SEND).\mathbf{nil}$

==========

The source's activity after it has already stored packets :

$Src(\text{s, d, apID, } e^s_{1-3}\text{, sID, earAtmp}) \stackrel{def}{=}$
$c_{sup}(x_{sq}).\ \{\ x_c^{act} \leq T_{act},\ x_c^{ear} \leq T_{ear}\ \} \triangleright (x_c^{act} \leq T_{act},\ x_c^{ear} \leq T_{ear}) \hookrightarrow$
$fwdDt(\text{s, d, apID, } e^s_{1-3}\text{, sID, } x_{sq})$
$[\,]\ c_{siACK}(x_{ack}).\ \{\ x_c^{act} \leq T_{act},\ x_c^{ear} \leq T_{ear}\ \} \triangleright (x_c^{act} \leq T_{act},\ x_c^{ear} \leq T_{ear}) \hookrightarrow$
$rcvACKS(\text{s, d, apID, } e^s_{1-3}\text{, sID, earAtmp, } x_{ack})$
$[\,]\ c_{siNACK}(x_{nack}).\ \{\ x_c^{act} \leq T_{act},\ x_c^{ear} \leq T_{ear}\ \} \triangleright (x_c^{act} \leq T_{act},\ x_c^{ear} \leq T_{ear}) \hookrightarrow$
$rcvNACKS(\text{s, d, apID, } e^s_{1-3}\text{, sID, earAtmp, } x_{nack})$
$[\,]\ \{\ x_c^{act} \leq T_{act},\ x_c^{ear} \leq T_{ear}\ \} \triangleright (x_c^{act} \leq T_{act},\ x_c^{ear} \leq T_{ear}) \hookrightarrow c_{sessionEND}(= SEND).\mathbf{nil}$
$[\,]\ (x_c^{act} \geq T_{act}) \hookrightarrow actTimeOut$
$[\,]\ (x_c^{ear} \geq T_{ear}) \hookrightarrow earTimeOut;$
==========

$fwdDt(\text{s, d, apID, } e^s_{1-3}\text{, sID, sq}) \stackrel{def}{=}$
$[e_1^s = E]\ let\ e_1^s = (s, d, apID, sID, sq)\ in\ nxtStp1(\text{s, d, apID, } e^s_{1-3}\text{, sID, sq})\ else$
$[e_2^s = E]\ let\ e_2^s = (s, d, apID, sID, sq)\ in\ nxtStp2(\text{s, d, apID, } e^s_{1-3}\text{, sID, sq})\ else$
$[e_3^s = E]\ let\ e_3^s = (s, d, apID, sID, sq)\ in\ nxtStp3(\text{s, d, apID, } e^s_{1-3}\text{, sID, sq})\ else\ \overline{c_{error}}\langle ERROR\rangle.$
$Src(\text{s, d, apID, } e^s_{1-3}\text{, sID, earAtmp})$
$[\,]\ c_{sessionEND}(= SEND).\mathbf{nil}$

$nxtStp1(\text{s, d, apID, } e^s_{1-3}\text{, sID, sq}) \stackrel{def}{=}$
$[e_2^s = E]\ checkAW(\text{s, d, apID, } e^s_{1-3}\text{, sID, sq})\ else$
$[e_3^s = E]\ checkAW(\text{s, d, apID, } e^s_{1-3}\text{, sID, sq})\ else$
$let\ (ear,\ rtx,\ earAtmp) = (1,\ 0,\ 1)\ in\ (\ \overline{c_{si}}\langle s, d, apID, sID, sq, ear, rtx\rangle.\ Src(\text{s, d, apID, } e^s_{1-3}\text{, sID, earAtmp})$

)
[ ] $c_{sessionEND}(= SEND)$.**nil**

nxtStp2(s, d, apID, e^s_{1-3}, sID, sq) $\stackrel{def}{=}$
[$e^s_3 = E$] *checkAW*(s, d, apID, e^s_{1-3}, sID, sq) *else*
let (ear, rtx, earAtmp) = (1, 0, 1) in (
$\overline{c_{si}}\langle s, d, apID, sID, sq, ear, rtx\rangle$. *Src*(s, d, apID, e^s_{1-3}, sID, earAtmp)
)
[ ] $c_{sessionEND}(= SEND)$.**nil**

nxtStp3(s, d, apID, e^s_{1-3}, sID, sq) $\stackrel{def}{=}$
let (ear, rtx, earAtmp) = (1, 0, 1) in (
$\overline{c_{si}}\langle s, d, apID, sID, sq, ear, rtx\rangle$. *Src*(s, d, apID, e^s_{1-3}, sID, earAtmp)
)
[ ] $c_{sessionEND}(= SEND)$.**nil**

---

checkAW(s, d, apID, e^s_{1-3}, sID, sq)$\stackrel{def}{=}$
[$sq = AW$] *let (ear, rtx, earAtmp) = (1, 0, 1) in*
(
$\overline{c_{si}}\langle s, d, apID, sID, sq, ear, rtx\rangle$. *Src*(s, d, apID, e^s_{1-3}, sID))
) *else*
(
let (ear, rtx) = (0,0) in
(
$\overline{c_{si}}\langle s, d, apID, sID, sq, ear, rtx\rangle$. *Src*(s, d, apID, e^s_{1-3}, sID, earAtmp)
)
)
[ ] $c_{sessionEND}(= SEND)$.**nil**

==========

rcvACKS(s, d, apID, e^s_{1-3}, sID, earAtmp)$\stackrel{def}{=}$
c_{siACK}(acknum). *hndleACK*(s, d, apID, e^s_{1-3}, sID, acknum)
[ ] $c_{sessionEND}(= SEND)$.**nil**

hndleACK(s, d, apID, e^s_{1-3}, sID, acknum)$\stackrel{def}{=}$
[$5(e^s_1) \leq acknum$] *checkE1*(s, d, apID, e^s_{1-3}, sID, acknum) *else*
[$5(e^s_2) \leq acknum$] *checkE2*(s, d, apID, e^s_{1-3}, sID, acknum) *else*
[$5(e^s_3) \leq acknum$] *checkE3*(s, d, apID, e^s_{1-3}, sID, acknum) *else*
let (earAtmp = 0) in $\parallel x^{act}_c \parallel$ *Src*(s, d, apID, e^s_{1-3}, sID, earAtmp)
[ ] $c_{sessionEND}(= SEND)$.**nil**

---

checkE1(s, d, apID, e^s_{1-3}, sID, acknum)$\stackrel{def}{=}$ *let* ($e^s_1 = E$) *in*
[$5(e^s_2) \leq acknum$] *checkE2*(s, d, apID, e^s_{1-3}, sID, acknum) *else*
[$5(e^s_3) \leq acknum$] *checkE3*(s, d, apID, e^s_{1-3}, sID, acknum) *else*

let (earAtmp = 0) in $\| \; x_c^{act} \; \|$ *Src*(s, d, apID, e^s_{1-3}, sID, earAtmp)
[ ] $c_{sessionEND}(= SEND)$.**nil**

checkE2(s, d, apID, e^s_{1-3}, sID, acknum) $\stackrel{def}{=}$ *let* (e^s_2 = *E) in*
[$5(e^s_3) \leq acknum$] *checkE3*(s, d, apID, e^s_{1-3}, sID, acknum) *else*
let (earAtmp = 0) in $\| \; x_c^{act} \; \|$ *Src*(s, d, apID, e^s_{1-3}, sID, earAtmp)
[ ] $c_{sessionEND}(= SEND)$.**nil**

checkE3(s, d, apID, e^s_{1-3}, sID, acknum) $\stackrel{def}{=}$ *let* (e^s_3 = *E) in*
let (earAtmp = 0) in $\| \; x_c^{act} \; \|$ *Src*(s, d, apID, e^s_{1-3}, sID, earAtmp)
[ ] $c_{sessionEND}(= SEND)$.**nil**

==========

rcvNACKS(s, d, apID, e^s_{1-3}, sID, earAtmp) $\stackrel{def}{=}$
c_{siNACK}(acknum,b1). *hndleACKNACKS1*(s, d, apID, e^s_{1-3}, sID, acknum, b1)
[ ] c_{siNACK}(acknum,b1,b2). *hndleACKNACKS2*(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2)
[ ] c_{siNACK}(acknum,b1,b2,b3). *hndleACKNACKS3*(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2, b3)
[ ] $c_{sessionEND}(= SEND)$.**nil**

hndleACKNACKS1(s, d, apID, e^s_{1-3}, sID, acknum, b1) $\stackrel{def}{=}$ *let* (e^s_1 = *E) in*
[$5(e^s_1) \leq acknum$] *checkE1Nck1*(s, d, apID, e^s_{1-3}, sID, acknum, b1) *else*
[$5(e^s_2) \leq acknum$] *checkE2Nck1*(s, d, apID, e^s_{1-3}, sID, acknum, b1) *else*
[$5(e^s_3) \leq acknum$] *checkE3Nck1*(s, d, apID, e^s_{1-3}, sID, acknum, b1) *else*
let (earAtmp = 0) in isSetLst(s, d, apID, e^s_{1-3}, sID, acknum, b1)
[ ] $c_{sessionEND}(= SEND)$.**nil**

checkE1Nck1(s, d, apID, e^s_{1-3}, sID, acknum, b1) $\stackrel{def}{=}$
[$5(e^s_2) \leq acknum$] *checkE2Nck1*(s, d, apID, e^s_{1-3}, sID, acknum, b1) *else*
[$5(e^s_3) \leq acknum$] *checkE3Nck1*(s, d, apID, e^s_{1-3}, sID, acknum, b1) *else*
let (earAtmp = 0) in isSetLst(s, d, apID, e^s_{1-3}, sID, acknum, b1)
[ ] $c_{sessionEND}(= SEND)$.**nil**

checkE2Nck1(s, d, apID, e^s_{1-3}, sID, acknum, b1) $\stackrel{def}{=}$ *let* (e^s_2 = *E) in*
[$5(e^s_3) \leq acknum$] *checkE3Nck1*(s, d, apID, e^s_{1-3}, sID, acknum, b1) *else*
let (earAtmp = 0) in isSetLst(s, d, apID, e^s_{1-3}, sID, acknum, b1)
[ ] $c_{sessionEND}(= SEND)$.**nil**

checkE3Nck1(s, d, apID, e^s_{1-3}, sID, acknum, b1) $\stackrel{def}{=}$ *let* (e^s_3, *earAtmp*) = (*E*, 0) *in*
isSetLst(s, d, apID, e^s_{1-3}, sID, acknum, b1)
[ ] $c_{sessionEND}(= SEND)$.**nil**

---

hndleACKNACKS2(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2) $\stackrel{def}{=}$
[$5(e^s_1) \leq acknum$] *checkE1Nck2*(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2) *else*

$[5(e_2^s) \leq acknum]$ *checkE2Nck2*(s, d, apID, e_{1-3}^s, sID, acknum, b1, b2) *else*
$[5(e_3^s) \leq acknum]$ *checkE3Nck2*(s, d, apID, e_{1-3}^s, sID, acknum, b1, b2) *else*
let (earAtmp = 0) in isSet(s, d, apID, e_{1-3}^s, sID, acknum, b1).
isSetLst(s, d, apID, e_{1-3}^s, sID, acknum, b2)
[ ] $c_{sessionEND}(= SEND)$.**nil**

checkE1Nck2(s, d, apID, e_{1-3}^s, sID, acknum, b1, b2) $\stackrel{def}{=}$ *let* $(e_1^s = E)$ *in*
$[5(e_2^s) \leq acknum]$ *checkE2Nck2*(s, d, apID, e_{1-3}^s, sID, acknum, b1, b2) *else*
$[5(e_3^s) \leq acknum]$ *checkE3Nck2*(s, d, apID, e_{1-3}^s, sID, acknum, b1, b2) *else*
let (earAtmp = 0) in isSet(s, d, apID, e_{1-3}^s, sID, acknum, b1).
isSetLst(s, d, apID, e_{1-3}^s, sID, acknum, b2)
[ ] $c_{sessionEND}(= SEND)$.**nil**

checkE2Nck2(s, d, apID, e_{1-3}^s, sID, acknum, b1, b2) $\stackrel{def}{=}$ *let* $(e_2^s = E)$ *in*
$[5(e_3^s) \leq acknum]$ *checkE3Nck1*(s, d, apID, e_{1-3}^s, sID, acknum, b1, b2) *else*
let (earAtmp = 0) in isSet(s, d, apID, e_{1-3}^s, sID, acknum, b1).
isSetLst(s, d, apID, e_{1-3}^s, sID, acknum, b2)
[ ] $c_{sessionEND}(= SEND)$.**nil**

checkE3Nck2(s, d, apID, e_{1-3}^s, sID, acknum, b1, b2) $\stackrel{def}{=}$ *let* $(e_3^s,\ earAtmp) = (E,\ 0)$ *in*
isSet(s, d, apID, e_{1-3}^s, sID, acknum, b1).*isSetLst*(s, d, apID, e_{1-3}^s, sID, acknum, b2)
[ ] $c_{sessionEND}(= SEND)$.**nil**

---

hndleACKNACKS3(s, d, apID, e_{1-3}^s, sID, acknum, b1, b2, b3) $\stackrel{def}{=}$
$[5(e_1^s) \leq acknum]$ *checkE1Nck3*(s, d, apID, e_{1-3}^s, sID, acknum, b1, b2, b3) *else*
$[5(e_2^s) \leq acknum]$ *checkE2Nck3*(s, d, apID, e_{1-3}^s, sID, acknum, b1, b2, b3) *else*
$[5(e_3^s) \leq acknum]$ *checkE3Nck3*(s, d, apID, e_{1-3}^s, sID, acknum, b1, b2, b3) *else*
let (earAtmp = 0) in isSet(s, d, apID, e_{1-3}^s, sID, acknum, b1).
isSet(s, d, apID, e_{1-3}^s, sID, acknum, b2).*isSetLst*(s, d, apID, e_{1-3}^s, sID, acknum, b3)
[ ] $c_{sessionEND}(= SEND)$.**nil**

checkE1Nck3(s, d, apID, e_{1-3}^s, sID, acknum, b1, b2, b3) $\stackrel{def}{=}$
let $(e_1^s = E)$ *in*
$[5(e_2^s) \leq acknum]$ *checkE2Nck3*(s, d, apID, e_{1-3}^s, sID, acknum, b1, b2, b3) *else*
$[5(e_3^s) \leq acknum]$ *checkE3Nck3*(s, d, apID, e_{1-3}^s, sID, acknum, b1, b2, b3) *else*
let (earAtmp = 0) in isSet(s, d, apID, e_{1-3}^s, sID, acknum, b1)
isSet(s, d, apID, e_{1-3}^s, sID, acknum, b2). *isSetLst*(s, d, apID, e_{1-3}^s, sID, acknum, b3)
[ ] $c_{sessionEND}(= SEND)$.**nil**

checkE2Nck3(s, d, apID, e_{1-3}^s, sID, acknum, b1, b2, b3) $\stackrel{def}{=}$
let $(e_2^s = E)$ *in*
$[5(e_3^s) \leq acknum]$ *checkE3Nck3*(s, d, apID, e_{1-3}^s, sID, acknum, b1, b2, b3) *else*
let (earAtmp = 0) in isSet(s, d, apID, e_{1-3}^s, sID, acknum, b1).
isSet(s, d, apID, e_{1-3}^s, sID, acknum, b2).*isSetLst*(s, d, apID, e_{1-3}^s, sID, acknum, b3)
[ ] $c_{sessionEND}(= SEND)$.**nil**

checkE3Nck3(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2, b3) $\stackrel{def}{=}$
let (e^s_3, *earAtmp*) = (*E*, 0) *in* *isSet*(s, d, apID, e^s_{1-3}, sID, acknum, b1).
isSet(s, d, apID, e^s_{1-3}, sID, acknum, b2). *isSetLst*(s, d, apID, e^s_{1-3}, sID, acknum, b3)
[ ] $c_{sessionEND}(= SEND)$.**nil**

---

isSet(s, d, apID, e^s_{1-3}, sID, acknum, bt) $\stackrel{def}{=}$
$[5(e^s_1) = bt]$ *rtxPck*(s, d, apID, e^s_{1-3}, sID, acknum, bt) *else*
$[5(e^s_2) = bt]$ *rtxPck*(s, d, apID, e^s_{1-3}, sID, acknum, bt) *else*
$[5(e^s_3) = bt]$ *rtxPck*(s, d, apID, e^s_{1-3}, sID, acknum, bt) *else* **nil**
[ ] $c_{sessionEND}(= SEND)$.**nil**

rtxPck(s, d, apID, e^s_{1-3}, sID, acknum, bt) $\stackrel{def}{=}$
$[5(e^s_1) = bt]$ *let* (*ear, rtx*) = (0,1) *in* $\overline{c_{si}}\langle s, d, apID, sID, bt, ear, rtx\rangle$.**nil**
[ ] $c_{sessionEND}(= SEND)$.**nil**

isSetLst(s, d, apID, e^s_{1-3}, sID, acknum, bt) $\stackrel{def}{=}$
$[5(e^s_1) = bt]$ *rtxPckLst*(s, d, apID, e^s_{1-3}, sID, acknum, bt) *else*
$[5(e^s_2) = bt]$ *rtxPckLst*(s, d, apID, e^s_{1-3}, sID, acknum, bt) *else*
$[5(e^s_3) = bt]$ *rtxPckLst*(s, d, apID, e^s_{1-3}, sID, acknum, bt) *else*
$\| x^{act}_c \|$ *Src*(s, d, apID, e^s_{1-3}, sID, earAtmp)
[ ] $c_{sessionEND}(= SEND)$.**nil**

rtxPckLst(s, d, apID, e^s_{1-3}, sID, acknum, bt) $\stackrel{def}{=}$
(*let* (*ear,rtx*) = (1,1) *in* $\overline{c_{si}}\langle s, d, apID, sID, bt, ear, rtx\rangle$.
$\| x^{act}_c \|$ *Src*(s, d, apID, e^s_{1-3}, sID, earAtmp)
[ ] $c_{sessionEND}(= SEND)$.**nil**

==========
==========

Process that models the (probabilistic) behavior of an intermediate node:
Int(e^i_{1-3}) $\stackrel{def}{=}$
$c_{si}(s, d, apID, sID, sq, ear, rtx)$.*hndleDtI*(s, d, apID, sID, sq, ear, rtx, e^i_{1-3})
[ ] *rcvACKI*(e^i_{1-3}) [ ] *rcvNACKI*(e^i_{1-3}) [ ] $c_{sessionEND}(= sEND)$.**nil**;

==========

/* Adding a probabilistic choice for storing a pck or not */
hndleDtI(s, d, apID, sID, sq, ear, rtx, e^i_{1-3}) $\stackrel{def}{=}$
$[e^i_1 = (s, d, apID, sID, sq)]$ $\overline{c_{id}}\langle s, d, apID, sID, bt, ear, rtx\rangle$.*Int*($e^i_{1-3}$) *else*
$[e^i_2 = (s, d, apID, sID, sq)]$ $\overline{c_{id}}\langle s, d, apID, sID, bt, ear, rtx\rangle$.*Int*($e^i_{1-3}$) *else*
$[e^i_3 = (s, d, apID, sID, sq)]$ $\overline{c_{id}}\langle s, d, apID, sID, bt, ear, rtx\rangle$.*Int*($e^i_{1-3}$) *else*
strAndFwI(s, d, apID, sID, sq, ear, rtx, e^i_{1-3}) $\oplus_p$ *FwI*(s, d, apID, sID, sq, ear, rtx, e^i_{1-3});

FwI(s, d, apID, sID, sq, ear, rtx, e^i_{1-3}) $\stackrel{def}{=}$ $\overline{c_{id}}\langle s, d, apID, sID, bt, ear, rtx\rangle$.*Int*($e^i_{1-3}$);

$strAndFwI$(s, d, apID, sID, sq, ear, rtx, e^i_{1-3}) $\stackrel{def}{=}$
$[e^i_1 = E]$ *let* $e^i_1 = (s, d, apID, sID, sq)$ *in* $\overline{c_{id}}\langle s, d, apID, sID, bt, ear, rtx\rangle.Int(e^i_{1-3})$ *else*
$[e^i_2 = E]$ *let* $e^i_2 = (s, d, apID, sID, sq)$ *in* $\overline{c_{id}}\langle s, d, apID, sID, bt, ear, rtx\rangle.Int(e^i_{1-3})$ *else*
$[e^i_3 = E]$ *let* $e^i_3 = (s, d, apID, sID, sq)$ *in* $\overline{c_{id}}\langle s, d, apID, sID, bt, ear, rtx\rangle.Int(e^i_{1-3})$ *else*
$\overline{c_{id}}\langle s, d, apID, sID, bt, ear, rtx\rangle.Int(e^i_{1-3})$;

==========

$rcvACKI(e^i_{1-3}) \stackrel{def}{=}$
c_{idACK}(acknum).hndleACKI(s, d, apID, e^i_{1-3}, sID, acknum);

$hndleACKI$(s, d, apID, e^i_{1-3}, sID, acknum) $\stackrel{def}{=}$
$[5(e^i_1) \leq acknum]$ $checkE1I$(s, d, apID, e^i_{1-3}, sID, acknum) *else*
$[5(e^i_2) \leq acknum]$ $checkE2I$(s, d, apID, e^i_{1-3}, sID, acknum) *else*
$[5(e^i_3) \leq acknum]$ $checkE3I$(s, d, apID, e^i_{1-3}, sID, acknum) *else* $\overline{c_{siACK}}\langle acknum\rangle.Int(e^i_{1-3})$;

$checkE1I$(s, d, apID, e^i_{1-3}, sID, acknum) $\stackrel{def}{=}$
let $(e^i_1 = E)$ *in* $[e^i_2 \leq acknum]$ $checkE2I$(s, d, apID, e^i_{1-3}, sID, acknum) *else*
$[5(e^i_3) \leq acknum]$ $checkE3I$(s, d, apID, e^i_{1-3}, sID, acknum) *else* $\overline{c_{siACK}}\langle acknum\rangle.Int(e^i_{1-3})$;

$checkE2I$(s, d, apID, e^i_{1-3}, sID, acknum) $\stackrel{def}{=}$
let $(e^i_2 = E)$ *in* $[5(e^i_3) \leq acknum]$ $checkE3I$(s, d, apID, e^i_{1-3}, sID, acknum) *else*
$\overline{c_{siACK}}\langle acknum\rangle.Int(e^i_{1-3})$;

$checkE3I$(s, d, apID, e^i_{1-3}, sID, acknum) $\stackrel{def}{=}$
let $(e^i_3 = E)$ *in* $\overline{c_{siACK}}\langle acknum\rangle.Int(e^i_{1-3})$;

==========

$rcvNACKI$(s, d, apID, e^i_{1-3}, sID, earAtmp)$\stackrel{def}{=}$
c_{idNACK}(acknum,b1).hndleACKNACKI1(s, d, apID, e^i_{1-3}, sID, acknum, b1)
[ ] c_{idNACK}(acknum,b1,b2).hndleACKNACKI2(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2)
[ ] c_{idNACK}(acknum,b1,b2,b3).hndleACKNACKI3(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2, b3);

$hndleACKNACKI1$(s, d, apID, e^i_{1-3}, sID, acknum, b1)$\stackrel{def}{=}$
$[5(e^i_1) \leq acknum]$ $checkE1NckI1$(s, d, apID, e^i_{1-3}, sID, acknum, b1) *else*
$[5(e^i_2) \leq acknum]$ $checkE2NckI1$(s, d, apID, e^i_{1-3}, sID, acknum, b1) *else*
$[5(e^i_3) \leq acknum]$ $checkE3NckI1$(s, d, apID, e^i_{1-3}, sID, acknum, b1) *else*
$hndleNACKI1$(s, d, apID, e^i_{1-3}, sID, acknum, b1);

$checkE1NckI1$(s, d, apID, e^i_{1-3}, sID, acknum, b1)$\stackrel{def}{=}$
let $(e^i_1 = E)$ *in* $[5(e^i_2) \leq acknum]$ $checkENck2I1$(s, d, apID, e^i_{1-3}, sID, acknum, b1) *else*
$[5(e^i_3) \leq acknum]$ $checkE3NckI1$(s, d, apID, e^i_{1-3}, sID, acknum, b1) *else*
$hndleNACKI1$(s, d, apID, e^i_{1-3}, sID, acknum, b1);

$checkE2NckI1$(s, d, apID, e^i_{1-3}, sID, acknum, b1)$\stackrel{def}{=}$
let $(e^i_2 = E)$ *in* $[5(e^i_3) \leq acknum]$ $checkE3NckI1$(s, d, apID, e^i_{1-3}, sID, acknum, b1) *else*
$hndleNACKI1$(s, d, apID, e^i_{1-3}, sID, acknum, b1);

$checkE3NckI1$(s, d, apID, e^i_{1-3}, sID, acknum, b1) $\stackrel{def}{=}$
let $(e^i_3 = E)$ *in* $hndleNACKI1$(s, d, apID, e^i_{1-3}, sID, acknum, b1);

==========

hndleNACKI1(s, d, apID, e^i_{1-3}, sID, acknum, bt) $\stackrel{def}{=}$
$[5(e^i_1) = bt]$ *rtxPckfwAck1*(s, d, apID, e^i_{1-3}, sID, acknum, bt) *else*
$[5(e^i_2) = bt]$ *rtxPckfwAck1*(s, d, apID, e^i_{1-3}, sID, acknum, bt) *else*
$[5(e^i_3) = bt]$ *rtxPckfwAck1*(s, d, apID, e^i_{1-3}, sID, acknum, bt) *else* $\overline{c_{siNACK}}\langle acknum, bt\rangle$.**nil**;

rtxPckfwAck1(s, d, apID, e^i_{1-3}, sID, acknum, bt) $\stackrel{def}{=}$
let (ear = 0) in let (rtx = 1) in $\overline{c_{id}}\langle s, d, apID, sID, bt, ear, rtx\rangle.\overline{c_{siACK}}\langle acknum\rangle$.**nil**;

==========

hndleACKNACKI2(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2) $\stackrel{def}{=}$
$[5(e^i_1) \leq acknum]$ *checkE1NckI2*(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2) *else*
$[5(e^i_2) \leq acknum]$ *checkE2NckI2*(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2) *else*
$[5(e^i_3) \leq acknum]$ *checkE3NckI2*(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2) *else*
hndleNACKI2(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2);

checkE1NckI2(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2) $\stackrel{def}{=}$
let $(e^i_1 = E)$ *in* $[5(e^i_2) \leq acknum]$ *checkENck2I2*(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2) *else*
$[5(e^i_3) \leq acknum]$ *checkE3NckI2*(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2) *else*
hndleNACKI2(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2);

checkE2NckI2(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2) $\stackrel{def}{=}$
let $(e^i_2 = E)$ *in* $[5(e^i_3) \leq acknum]$ *checkE3NckI2*(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2) *else*
hndleNACKI2(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2);

checkE3NckI2(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2) $\stackrel{def}{=}$
let $(e^i_3 = E)$ *in hndleNACKI2*(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2);

==========

hndleNACKI2(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2) $\stackrel{def}{=}$
$[5(e^i_1) = b1]$ *let (ear = 0) in let (rtx = 1) in* $\overline{c_{id}}\langle s, d, apID, sID, b1, ear, rtx\rangle.$
b1YesNxtb2(s, d, apID, e^i_{1-3}, sID, acknum, b2) *else*
$[5(e^i_2) = b1]$ *let (ear = 0) in let (rtx = 1) in* $\overline{c_{id}}\langle s, d, apID, sID, b1, ear, rtx\rangle.$
b1YesNxtb2(s, d, apID, e^i_{1-3}, sID, acknum, b2) *else*
$[5(e^i_3) = b1]$ *let (ear = 0) in let (rtx = 1) in* $\overline{c_{id}}\langle s, d, apID, sID, b1, ear, rtx\rangle.$
b1YesNxtb2(s, d, apID, e^i_{1-3}, sID, acknum, b2) *else*
b1NotNxtb2(s, d, apID, e^i_{1-3}, sID, acknum, b2);

b1YesNxtb2(s, d, apID, e^i_{1-3}, sID, acknum, b2) $\stackrel{def}{=}$
$[5(e^i_1) = b2]$ *b1Yesb2Yes*(s, d, apID, e^i_{1-3}, sID, acknum, b2) *else*
$[5(e^i_2) = b2]$ *b1Yesb2Yes*(s, d, apID, e^i_{1-3}, sID, acknum, b2) *else*
$[5(e^i_3) = b2]$ *b1Yesb2Yes*(s, d, apID, e^i_{1-3}, sID, acknum, b2) *else* $\overline{c_{siNACK}}\langle acknum, b2\rangle$.**nil**;

b1Yesb2Yes(s, d, apID, e^i_{1-3}, sID, acknum, bt) $\stackrel{def}{=}$
let (ear = 0) in let (rtx = 1) in $\overline{c_{id}}\langle s, d, apID, sID, bt, ear, rtx\rangle.\overline{c_{siACK}}\langle acknum\rangle$.**nil**;

b1NotNxtb2(s, d, apID, e^i_{1-3}, sID, acknum, b2) $\stackrel{def}{=}$
$[5(e^i_1) = b2]$ *b1Notb2Yes*(s, d, apID, e^i_{1-3}, sID, acknum, b2) *else*
$[5(e^i_2) = b2]$ *b1Notb2Yes*(s, d, apID, e^i_{1-3}, sID, acknum, b2) *else*
$[5(e^i_3) = b2]$ *b1Notb2Yes*(s, d, apID, e^i_{1-3}, sID, acknum, b2) *else* $\overline{c_{siNACK}}\langle acknum, b1, b2\rangle$.**nil**;

b1Notb2Yes(s, d, apID, e^i_{1-3}, sID, acknum, bt) $\stackrel{def}{=}$
let (ear = 0) in let (rtx = 1) in $\overline{c_{id}}\langle s, d, apID, sID, bt, ear, rtx\rangle.\overline{c_{siNACK}}\langle acknum, b1\rangle$.**nil**;

==========

hndleACKNACKI3(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2, b3) $\stackrel{def}{=}$
$[5(e^i_1) \leq acknum]$ *checkE1NckI3*(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2, b3) *else*
$[5(e^i_2) \leq acknum]$ *checkE2NckI3*(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2, b3) *else*
$[5(e^i_3) \leq acknum]$ *checkE3NckI3*(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2, b3) *else*
hndleNACKI3(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2, b3);

The functions *checkE1NckI3*, *checkE2NckI3*, *checkE3NckI3* and *hndleNACKI3* are specified in the same concept as the functions *checkE1NckI3*, *checkE2NckI3*, and *hndleNACKI2*. Next we turn to specify the behavior of the destination node.

==========
==========

Process that models the behavior of the destination:
Dst(e^d_{1-3}, ackNbr, nackNbr, toRTX1, nxtsq) $\stackrel{def}{=}$
$c_{id}(s, d, apID, sID, sq, ear, rtx)$.*hndleDtDst* [ ] $c_{sessionEND}(= sEND)$.**nil**;

hndleDtDst $\stackrel{def}{=}$
/* Duplicated → check if EAR bit is set → missing packets */
$[e^d_1 = (s, d, apID, sID, sq)]$ *checkEARis1 else*
$[e^d_2 = (s, d, apID, sID, sq)]$ *checkEARis1 else*
/* NOT duplicated → store → fw to uplayer in-seq packets */
$[e^d_3 = (s, d, apID, sID, sq)]$ *checkEARis1 else strAndFwDst*;

checkEARis1 $\stackrel{def}{=}$
$[ear = 1]$ *sndACKNACKDst else Dst*(e^d_{1-3}, ackNbr, nackNbr, toRTX1, nxtsq);

sndACK1NACKDst $\stackrel{def}{=}$
/* if nackNbr = 0 snd ACK, if nackNbr > 0 snd NACK*/
$[nackNbr = 0]$ $\overline{c_{idACK}}\langle ackNbr\rangle$.*Dst*($e^d_{1-3}$, ackNbr, nackNbr, toRTX1, nxtsq) *else*
$[nackNbr = 1]$ *1BitInNACK else*
$[nackNbr = 2]$ *2BitInNACK else*
/* We allow this case for modelling the attacker ability to set sq in EAR to 4. */
$[nackNbr = 3]$ $\overline{c_{idNACK}}\langle ackNbr, 1, 2, 3\rangle$.*Dst*($e^d_{1-3}$, ackNbr, nackNbr, toRTX1, nxtsq);

1BitInNACK $\stackrel{def}{=}$
/* Assume dst knows it should get: 1, 2, 3. Hence num of NACK bits ≤ 3 */
$[sq = 2]$ $\overline{c_{idNACK}}\langle ackNbr, 1\rangle$.*Dst*($e^d_{1-3}$, ackNbr, nackNbr, toRTX1, nxtsq) *else*
$[sq = 3]$ *let sqnm=1 in checkMissingPcktsSQE3 else*
$[sq > 3]$ *let sqnm=1 in checkMissingPcktsSQG3;*

checkMissingPcktsSQE3 $\stackrel{def}{=}$
$[sqnm \leq 2]$ *goOnMissingPcktsSQE3 else Dst*(e^d_{1-3}, ackNbr, nackNbr, toRTX1, nxtsq);

$goOnMissingPcktsSQE3 \stackrel{def}{=}$
$[5(e_1^d) = sqnm]\ \overline{c_{idNACK}}\langle ackNbr, sqnm\rangle.Dst(e_{1-3}^d$, ackNbr, nackNbr, toRTX1, nxtsq) *else*
$[5(e_2^d) = sqnm]\ \overline{c_{idNACK}}\langle ackNbr, sqnm\rangle.Dst(e_{1-3}^d$, ackNbr, nackNbr, toRTX1, nxtsq) *else*
$[5(e_3^d) = sqnm]\ \overline{c_{idNACK}}\langle ackNbr, sqnm\rangle.Dst(e_{1-3}^d$, ackNbr, nackNbr, toRTX1, nxtsq) *else*
let sqnm=incr(sqnm) in checkMissingPcktsSQE3;

$checkMissingPcktsSQG3 \stackrel{def}{=}$
$[sqnm \leq 3]\ goOnMissingPcktsSQG3\ else\ Dst(e_{1-3}^d$, ackNbr, nackNbr, toRTX1, nxtsq);

$goOnMissingPcktsSQG3 \stackrel{def}{=}$
$[5(e_1^d) = sqnm]\ \overline{c_{idNACK}}\langle ackNbr, sqnm\rangle.Dst(e_{1-3}^d$, ackNbr, nackNbr, toRTX1, nxtsq) *else*
$[5(e_2^d) = sqnm]\ \overline{c_{idNACK}}\langle ackNbr, sqnm\rangle.Dst(e_{1-3}^d$, ackNbr, nackNbr, toRTX1, nxtsq) *else*
$[5(e_3^d) = sqnm]\ \overline{c_{idNACK}}\langle ackNbr, sqnm\rangle.Dst(e_{1-3}^d$, ackNbr, nackNbr, toRTX1, nxtsq) *else*
let sqnm=incr(sqnm) in checkMissingPcktsSQG3;

$2BitInNACK \stackrel{def}{=}$
/* Assume dst knows it should get: 1, 2, 3. Hence num of NACK bits ≤ 3 */
$[sq = 3]\ \overline{c_{idNACK}}\langle ackNbr, 1, 2\rangle.Dst(e_{1-3}^d$, ackNbr, nackNbr, toRTX1, nxtsq) *else*
$[sq > 3]$ *let sqnm=1 in check2BitMissingPckts1stBit*;

$check2BitMissingPckts1stBit \stackrel{def}{=}$
$[sqnm > 3]\ Dst(e_{1-3}^d$, ackNbr, nackNbr, toRTX1, nxtsq) *else*
$[5(e_1^d) = sqnm]$ *let toRTX1=sqnm in let sqnm=incr(sqnm) in goOn2BitMissingPckts2ndBit else*
$[5(e_2^d) = sqnm]$ *let toRTX1=sqnm in let sqnm=incr(sqnm) in goOn2BitMissingPckts2ndBit else*
$[5(e_3^d) = sqnm]$ *let toRTX1=sqnm in let sqnm=incr(sqnm) in goOn2BitMissingPckts2ndBit else*
let sqnm=incr(sqnm) in goOn2BitMissingPckts1stBit;

$check2BitMissingPckts2ndBit \stackrel{def}{=}$
$[sqnm > 3]\ Dst(e_{1-3}^d$, ackNbr, nackNbr, toRTX1, nxtsq) *else*
$[5(e_1^d) = sqnm]\ \overline{c_{idNACK}}\langle ackNbr, toRTX1, sqnm\rangle.$
$Dst(e_{1-3}^d$, ackNbr, nackNbr, toRTX1, nxtsq) *else*
$[5(e_2^d) = sqnm]\ \overline{c_{idNACK}}\langle ackNbr, toRTX1, sqnm\rangle.$
$Dst(e_{1-3}^d$, ackNbr, nackNbr, toRTX1, nxtsq) *else*
$[5(e_3^d) = sqnm]\ \overline{c_{idNACK}}\langle ackNbr, toRTX1, sqnm\rangle.$
$Dst(e_{1-3}^d$, ackNbr, nackNbr, toRTX1, nxtsq) *else*
let sqnm=incr(sqnm) in goOn2BitMissingPckts2ndBit;

$strAndFwDst \stackrel{def}{=}$
/* If in-seq (i.e., nxtsq = sq) → store and fw to upLayer, else only store */
$[sq = nxtsq]\ strSndUp\ else\ [sq > nxtsq]\ strOnly$;

$strSndUp \stackrel{def}{=}$
/* Store and send to upLayer */
$[e_1^d = E]$ *let* $e_1^d = (s, d, apID, sID, sq)$ *in* $\overline{c_{dup}}\langle s, d, apID, sID, sq\rangle.$
/* Decrease nackNbr, update ackNbr, increase nxtsq, finally check the EAR bit */
let nackNbr = dec(nackNbr) in let ackNbr = nxtsq in let nxtsq = incr(nxtsq) in
checkEARis1 else $[e_2^d = E]$ *let* $e_2^d = (s, d, apID, sID, sq)$ *in* $\overline{c_{dup}}\langle s, d, apID, sID, sq\rangle.$
let nackNbr = dec(nackNbr) in let ackNbr = nxtsq in let nxtsq = incr(nxtsq) in
checkEARis1 else $[e_3^d = E]$ *let* $e_3^d = (s, d, apID, sID, sq)$ *in* $\overline{c_{dup}}\langle s, d, apID, sID, sq\rangle.$
let nackNbr = dec(nackNbr) in let ackNbr = nxtsq in let nxtsq = incr(nxtsq) in
checkEARis1;

$strOnly \stackrel{def}{=}$
/* Store and increase nackNbr, finally check EAR bit */
$[e_1^d = E]$ *let* $e_1^d = (s, d, apID, sID, sq)$ *in let nackNbr = incr(nackNbr) in checkEARis1 else*
$[e_2^d = E]$ *let* $e_2^d = (s, d, apID, sID, sq)$ *in let nackNbr = incr(nackNbr) in checkEARis1 else*
$[e_3^d = E]$ *let* $e_3^d = (s, d, apID, sID, sq)$ *in let nackNbr = incr(nackNbr) in checkEARis1*;

B The detailed specification of SDTP in $\boldsymbol{crypt}_{time}^{prob}$

B.1 Real SDTP version

To model the SDTP protocol we extend the specification of the DTSN protocol in the following way. First, the source node extends each packet with an ACK MAC and a NACK MAC, which is accomplished by modifying the subprocesses *nxtStp1*, *nxtStp2*, *nxtStp3* and *checkAW* within process *fwdDt* of *Src*.

nxtStp1(s, d, apID, e_{1-3}^s, sID, sq) $\stackrel{def}{=}$
$[e_2^s = E]$ *checkAW*(s, d, apID, e_{1-3}^s, sID, sq) *else*
$[e_3^s = E]$ *checkAW*(s, d, apID, e_{1-3}^s, sID, sq) *else*
let ear=1 in let rtx=0 in let earAtmp=1 in
let Kack = K(sq, ACK) in let Knack = K(sq, NACK) in
let ACKMAC = MAC((s, d, apID, sID, sq, ear, rtx), Kack) in
let NACKMAC = MAC((s, d, apID, sID, sq, ear, rtx), Knack) in
$\overline{c_{si}}\langle s, d, apID, sID, sq, ear, rtx, ACKMAC, NACKMAC\rangle$. *Src*(s, d, apID, e_{1-3}^s, sID, earAtmp);

nxtStp2(s, d, apID, e_{1-3}^s, sID, sq) $\stackrel{def}{=}$
$[e_3^s = E]$ *checkAW*(s, d, apID, e_{1-3}^s, sID, sq) *else*
let ear=1 in let rtx=0 in let earAtmp=1 in
let Kack = K(sq, ACK) in let Knack = K(sq, NACK) in
let ACKMAC = MAC((s, d, apID, sID, sq, ear, rtx), Kack) in
let NACKMAC = MAC((s, d, apID, sID, sq, ear, rtx), Knack) in
$\overline{c_{si}}\langle s, d, apID, sID, sq, ear, rtx, ACKMAC, NACKMAC\rangle$. *Src*(s, d, apID, e_{1-3}^s, sID, earAtmp);

nxtStp3(s, d, apID, e_{1-3}^s, sID, sq) $\stackrel{def}{=}$
let ear=1 in let rtx=0 in let earAtmp=1 in
let Kack = K(sq, ACK) in let Knack = K(sq, NACK) in
let ACKMAC = MAC((s, d, apID, sID, sq, ear, rtx), Kack) in
let NACKMAC = MAC((s, d, apID, sID, sq, ear, rtx), Knack) in
$\overline{c_{si}}\langle s, d, apID, sID, sq, ear, rtx, ACKMAC, NACKMAC\rangle$. *Src*(s, d, apID, e_{1-3}^s, sID, earAtmp);

checkAW(s, d, apID, e_{1-3}^s, sID, sq) $\stackrel{def}{=}$
$[sq = AW]$ (*let ear=1 in let rtx=0 in let earAtmp=1 in*
let Kack = K(sq, ACK) in let Knack = K(sq, NACK) in
let ACKMAC = MAC((s, d, apID, sID, sq, ear, rtx), Kack) in
let NACKMAC = MAC((s, d, apID, sID, sq, ear, rtx), Knack) in
$\overline{c_{si}}\langle s, d, apID, sID, sq, ear, rtx, ACKMAC, NACKMAC\rangle$.*Src*(s, d, apID, e_{1-3}^s, sID)) *else*
(
let ear=0 in let rtx=0 in
let Kack = K(sq, ACK) in let Knack = K(sq, NACK) in
let ACKMAC = MAC((s, d, apID, sID, sq, ear, rtx), Kack) in
let NACKMAC = MAC((s, d, apID, sID, sq, ear, rtx), Knack) in
$\overline{c_{si}}\langle s, d, apID, sID, sq, ear, rtx, ACKMAC, NACKMAC\rangle$.
Src(s, d, apID, e_{1-3}^s, sID, earAtmp)
);

In addition, we have to extend the $crypt^{prob}_{time}$ specification of DSTN with the verification of ACK MACs and NACK MACs when the source receives ACK and NACK packets. Formally, we extend the processes

$rcvACKS$(s, d, apID, e^s_{1-3}, sID, earAtmp) $\stackrel{def}{=}$
c_{siACK}(acknum, ackkey, nackkey).hndleACK(s, d, apID, e^s_{1-3}, sID, acknum, ackkey, nackkey);

the expected data on channel c_{siACK} is extended by ackkey, nackkey that represent per-packet ACK and NACK keys, which are also included as the parameters of process *hndleACK* and its sub-processes *checkE1*, *checkE2* and *checkE3*.

$hndleACK$(s, d, apID, e^s_{1-3}, sID, acknum, ackkey, nackkey) $\stackrel{def}{=}$
$[5(e^s_1) \leq acknum]$ $[CheckMac(6(e^s_1), ackkey) = ok]$
checkE1(s, d, apID, e^s_{1-3}, sID, acknum, ackkey, nackkey) *else*
$[5(e^s_2) \leq acknum]$ $[CheckMac(6(e^s_2), ackkey) = ok]$
checkE2(s, d, apID, e^s_{1-3}, sID, acknum, ackkey, nackkey) *else*
$[5(e^s_3) \leq acknum]$ $[CheckMac(6(e^s_3), ackkey) = ok]$
checkE3(s, d, apID, e^s_{1-3}, sID, acknum, ackkey, nackkey) *else*
let (earAtmp = Null) in Src(s, d, apID, e^s_{1-3}, sID, earAtmp);

we extend the specification of *hndleACK* with the verification of the stored ACK MAC using the keys included in the received ACK packets. This is modelley by the if construct in $crypt^{prob}_{time}$: $[CheckMac(6(e^s_i), ackkey) = ok]$. In particular, $CheckMac(6(e^s_i), ackkey)$ is the verification of ACK MAC, which is stored in the 6-th place in the cache entry e^s_i. The same extension is applied in processes *checkE1*, *checkE2* and *checkE3*.

When a NACK packet has been received the SDTP protocol includes verification of ACK MAC and NACK MACs. The structure of the NACK packet compared to DTSN case is extended with an ACK key (if any) and some NACK keys depending on the number of bits in the NACK packet. Hence, the expected data on channel c_{siNACK} is extended with the ackkey and nackkey parameters, for instance, instead of c_{siNACK}(acknum, b1) we have c_{siNACK}(acknum, b1, ackkey, nackkey1). Each process *hndleACKNACKSi* and *checkEiNckj* includes the received ACK key and NACK keys as process parameters. Namely, the verification part $[5(e^s_i) \leq acknum]$ is extended with $[CheckMac(6(e^s_i), ackkey) = ok]$ $[CheckMac(7(e^s_i), nackkey) = ok]$ for each $i \in \{1, 2, 3\}$.

The parameters of processes *isSet*(s, d, apID, e^s_{1-3}, sID, acknum, b) and *isSetLst*(s, d, apID, e^s_{1-3}, sID, acknum, b) are extended with the corresponding ackkey and nackkey. Finally, processes *rtxPck* and *rtxPckLst* are modified as follows:

$rtxPck$(s, d, apID, e^s_{1-3}, sID, acknum, bt) $\stackrel{def}{=}$
let Kack = K(sq, ACK) in let Knack = K(sq, NACK) in
let ACKMAC = MAC((s, d, apID, sID, sq, ear, rtx), Kack) in
let NACKMAC = MAC((s, d, apID, sID, sq, ear, rtx), Knack) in
let (ear = 0) in let (rtx = 1) in $\overline{c_{si}}\langle s, d, apID, sID, bt, ear, rtx, ACKMAC, NACKMAC\rangle$.**nil**;

and

$rtxPckLst$(s, d, apID, e^s_{1-3}, sID, acknum, bt) $\stackrel{def}{=}$
let Kack = K(sq, ACK) in let Knack = K(sq, NACK) in
let ACKMAC = MAC((s, d, apID, sID, sq, ear, rtx), Kack) in
let NACKMAC = MAC((s, d, apID, sID, sq, ear, rtx), Knack) in
let (ear = 1) in let (rtx = 1) in $\overline{c_{si}}\langle s, d, apID, sID, bt, ear, rtx, ACKMAC, NACKMAC\rangle$.
Src(s, d, apID, e^s_{1-3}, sID, earAtmp);

Now we turn to modifying process $Int(e^i_{1-3})$ according to the definition of the SDTP protocol. Process *hndleDtI* beside the parameters defined in case of DTSN, also includes ackmac and nackmac. Each cache entry at intermediate nodes stores the packets that contains an ACK MAC and NACK MAC at the 6-th and 7-th places, respectively.

$$
\begin{aligned}
&\mathit{hndleDtI}(\text{s, d, apID, sID, sq, ear, rtx, } e^i_{1-3}\text{, ackmac, nackmac}) \stackrel{def}{=}\\
&\quad [e^i_1 = (\text{s, d, apID, sID, sq, ackmac, nackmac})]\\
&\quad \overline{c_{id}}\langle \mathit{s,\ d,\ apID,\ sID,\ bt,\ ear,\ rtx,\ ackmac,\ nackmac}\rangle.\mathit{Int}(e^i_{1-3})\ \mathit{else}\\
&\quad [e^i_2 = (\text{s, d, apID, sID, sq, ackmac, nackmac})]\\
&\quad \overline{c_{id}}\langle \mathit{s,\ d,\ apID,\ sID,\ bt,\ ear,\ rtx,\ ackmac,\ nackmac}\rangle.\mathit{Int}(e^i_{1-3})\ \mathit{else}\\
&\quad [e^i_3 = (\text{s, d, apID, sID, sq, ackmac, nackmac})]\\
&\quad \overline{c_{id}}\langle \mathit{s,\ d,\ apID,\ sID,\ bt,\ ear,\ rtx,\ ackmac,\ nackmac}\rangle.\mathit{Int}(e^i_{1-3})\ \mathit{else}\\
&\quad \mathit{strAndFwI}(\text{s, d, apID, sID, sq, ear, rtx, } e^i_{1-3}\text{, ackmac, nackmac});
\end{aligned}
$$

In process *strAndFwI* each data packet is stored in the cache entry and then forwarded to the next node:

$$
\begin{aligned}
&\mathit{strAndFwI}(\text{s, d, apID, sID, sq, ear, rtx, } e^i_{1-3}\text{, ackmac, nackmac}) \stackrel{def}{=}\\
&\quad [e^i_1 = E]\ \mathit{let}\ e^i_1 = (\mathit{s,\ d,\ apID,\ sID,\ sq,\ ackmac,\ nackmac})\\
&\quad \mathit{in}\ \overline{c_{id}}\langle \mathit{s,\ d,\ apID,\ sID,\ bt,\ ear,\ rtx,\ ackmac,\ nackmac}\rangle.\mathit{Int}(e^i_{1-3})\ \mathit{else}\\
&\quad [e^i_2 = E]\ \mathit{let}\ e^i_2 = (\mathit{s,\ d,\ apID,\ sID,\ sq,\ ackmac,\ nackmac})\\
&\quad \mathit{in}\ \overline{c_{id}}\langle \mathit{s,\ d,\ apID,\ sID,\ bt,\ ear,\ rtx,\ ackmac,\ nackmac}\rangle.\mathit{Int}(e^i_{1-3})\ \mathit{else}\\
&\quad [e^i_3 = E]\ \mathit{let}\ e^i_3 = (\mathit{s,\ d,\ apID,\ sID,\ sq,\ ackmac,\ nackmac})\\
&\quad \mathit{in}\ \overline{c_{id}}\langle \mathit{s,\ d,\ apID,\ sID,\ bt,\ ear,\ rtx,\ ackmac,\ nackmac}\rangle.\mathit{Int}(e^i_{1-3})\ \mathit{else}\\
&\quad \overline{c_{id}}\langle \mathit{s,\ d,\ apID,\ sID,\ bt,\ ear,\ rtx,\ ackmac,\ nackmac}\rangle.\mathit{Int}(e^i_{1-3});
\end{aligned}
$$

in process *rcvACKI* the ACK message format required to be received on channel c_{idACK} is *(acknum, ackkey)*, and process *hndleACKI* contains ackkey as its parameter.

$$
\begin{aligned}
&\mathit{rcvACKI}(e^i_{1-3}) \stackrel{def}{=}\\
&\quad c_{idACK}(\text{acknum, ackkey}).\text{hndleACKI(s, d, apID, } e^i_{1-3}\text{, sID, acknum, ackkey});
\end{aligned}
$$

The specifications of processes *hndleACKI*, *checkE1I*, *checkE2I* and *checkE3I* are modified similarly as in case of processes *hndleACK* and its sub-processes *checkE1*, *checkE2* and *checkE3* of the *Src* process.

Regarding process *rcvNACKI*(s, d, apID, e^i_{1-3}, sID, earAtmp) that specifies the behavior when an intermediate node receives a NACK message, the expected messsage format on channel c_{idNACK} is the format of NACK messages, e.g., $c_{idNACK}(\mathit{acknum,b1})$ is modified to $c_{idNACK}(\mathit{acknum,\ b1,\ ackkey,\ nackkey})$. Processes *hndleACKNACKIi* for $i \in \{1, 2, 3\}$ are extended with paramemeters ackkey and a given number of nackkey.

Finally, the process *Dst* for the destination node is modified such that an SDTP packet is expected on channel c_{id}, namely, $c_{id}(\mathit{s,\ d,\ apID,\ sID,\ sq,\ ear,\ rtx,\ ackmac,\ nackmac})$. In process *hndleDtDst* the comparison $[e^d_i = (\mathit{s,\ d,\ apID,\ sID,\ sq,\ ackmac,\ nackmac})]$ is changed to $[e^d_i = (\mathit{s,\ d,\ apID,\ sID,\ sq,\ ackmac,\ nackmac})]$. One most important change resulted from the specification of the SDTP is that the destination node includes ACK and NACK keys in the ACK and NACK messages. These relevant changes is made in the process *sndACK1NACKDst*.

$$
\begin{aligned}
&\mathit{sndACK1NACKDst} \stackrel{def}{=}\\
&\textbf{\textit{/* if nackNbr = Null snd ACK, if nackNbr > 0 snd NACK*/}}\\
&\quad [\mathit{nackNbr} = 0]\ \mathit{let\ Kack = K(ackNbr,\ ACK)\ in}\ \overline{c_{idACK}}\langle \mathit{ackNbr,\ Kack}\rangle.\\
&\qquad\qquad \mathit{Dst}(e^d_{1-3}\text{, ackNbr, nackNbr, toRTX1, nxtsq})\ \mathit{else}\\
&\quad [\mathit{nackNbr} = 1]\ \mathit{1BitInNACK}\ \mathit{else}\\
&\quad [\mathit{nackNbr} = 2]\ \mathit{2BitInNACK}\ \mathit{else}
\end{aligned}
$$

/* We allow this case for modelling the attacker ability to set sq in EAR to 4. */
$[nackNbr = 3]$ *let Kack = K(ackNbr, ACK) in let Knack1 = K(1, NACK) in*
let Knack2 = K(2, NACK) in let Knack3 = K(3, NACK) in
$\overline{c_{idNACK}}\langle$*ackNbr, 1, 2, 3, Kack, Knack1, Knack2, Knack3*$\rangle$.
$Dst(e^d_{1-3}$, ackNbr, nackNbr, toRTX1, nxtsq);

The let constructs of type *let Kack = K(ackNbr, ACK) in* and *let Knack1 = K(1, NACK) in* are used to represent that the destination generates the ACK and NACK keys for ACK and NACK messages, respectively. The rest two cases where there is one bit or there are two bits in NACK message, represented by processes *1BitInNACK* and *2BitInNACK* are modified in the same way.

B.2 Ideal SDTP version

We introduce the new syntax sugar of *cryptcal* to shorten the specification of complex systems and protocols.

(R1) $[t_1 = t_2]$ ***or*** $[t_3 = t_4]$ P ***else*** Q;
(R2) ***let*** $(x_1, x_2, \ldots, x_n) = (t_1, t_2, \ldots, t_n)$ ***in*** P

The first rule (R1) is the syntax sugar for $[t_1 = t_2]$ P *else* ($[t_3 = t_4]$ P *else* Q), and the second rule (R2) is the shorthand for process *let* $(x_1 = t_1)$ *in let* $(x_2 = t_2)$ *in* ... *let* $(x_n = t_n)$ *in* P.

In the following, we discuss how the operation of the ideal version differs from the real version of the intermediate node. In the ideal version, the intermediate node I is informed about the message it should receive from S and D, whenever they sent a message to I. This kind of hidden channel communication is performed via private channels c_{privSI} and c_{privID}. Below we denote variables x^{rcv}_{should} and x^{rcv}_{really} to represent the message should be received and the one that I actually received, respectively.

$$c_{privSI}(x^{rcv}_{should}).c_{si}(x^{rcv}_{really}).[x^{rcv}_{should} = x^{rcv}_{really}]\ procHndleMsg;$$

$$c_{privID}(x^{rcv}_{should}).c_{id}(x^{rcv}_{really}).[x^{rcv}_{should} = x^{rcv}_{really}]\ procHndleMsg;$$

where *procHndleMsg* represents the process in which I continues to handle the received message (either a data packet or a control message) in the similar way as in the real version. More specifically, *procHndleMsg* can be *hndleDtI*(s, d, apID, sID, sq, ear, rtx, $e^i_1 - 3$), *hndleACKI*(s, d, apID, e^i_{1-3}, sID, acknum), or *hndleACKNACKI1*(s, d, apID, e^i_{1-3}, sID, acknum, b1), *hndleACKNACKI2*(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2), *hndleACKNACKI3*(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2, b3).

For the ideal version of SDTP, beside the extension given in case of DTSN, the specification of node I also includes the scenarios related to the verification of ACK MAC and NACK MAC. Whenever the verification of the MACs fails, the constant *BadControl* is output on the public channel c_{badpck}. Namely,

$rcvACKI(e^i_{1-3}) \stackrel{def}{=}$
c_{idACK}(acknum, ackkey).
$[CheckMac(6(e^s_1), ackkey) = ok]$ ***or*** $[CheckMac(6(e^s_2), ackkey) = ok]$ ***or***
$[CheckMac(6(e^s_3), ackkey) = ok]$ *hndleACKI*(s, d, apID, e^i_{1-3}, sID, acknum) ***else***
$\overline{c_{badpck}}\langle$BadControl$\rangle$.*Int*($e^i_{1-3}$);

and similarly when I receives a NACK message

rcvNACKI(s, d, apID, e^i_{1-3}, sID, earAtmp) $\stackrel{def}{=}$
c_{idNACK}(acknum, b1, ackkey, nackkeyb1).$[CheckMac(6(e^s_1), ackkey) = ok]$ ***or***
$[CheckMac(6(e^s_2), ackkey) = ok]$ ***or*** $[CheckMac(6(e^s_3), ackkey) = ok]$
checkNACKMACKIb1 ***else*** $\overline{c_{badpck}}\langle$BadControl$\rangle$.*Int*($e^i_{1-3}$);

[ ]
c_{idNACK}(acknum, b1, b2, ackkey, nackkeyb1, nackkeyb2).[$CheckMac(6(e_1^s), ackkey) = ok$] ***or***
[$CheckMac(6(e_2^s), ackkey) = ok$] ***or*** [$CheckMac(6(e_3^s), ackkey) = ok$]
checkNACKMACKIb1b2 ***else*** $\overline{c_{badpck}}\langle$BadControl$\rangle$.*Int(*$e_{1-3}^i$*)*;
[ ]
c_{idNACK}(acknum,b1, b2, b3, ackkey, nackkeyb1, nackkeyb2, nackkeyb3).
[$CheckMac(6(e_1^s), ackkey) = ok$] ***or*** [$CheckMac(6(e_2^s), ackkey) = ok$] ***or***
[$CheckMac(6(e_3^s), ackkey) = ok$]
checkNACKMACKIb1b2b3 ***else*** $\overline{c_{badpck}}\langle$BadControl$\rangle$.*Int(*$e_{1-3}^i$*)*;

checkNACKMACKIb1 $\stackrel{def}{=}$
[$CheckMac(7(e_1^s), nackkeyb1) = ok$] ***or*** [$CheckMac(7(e_2^s), nackkeyb1) = ok$] ***or***
[$CheckMac(7(e_3^s), nackkeyb1) = ok$] *hndleACKNACKI1*(s, d, apID, e_{1-3}^i, sID, acknum, b1)
else $\overline{c_{badpck}}\langle$BadControl$\rangle$.*Int(*$e_{1-3}^i$*)*;

checkNACKMACKIb1b2 $\stackrel{def}{=}$
[$CheckMac(7(e_1^s), nackkeyb1) = ok$] ***or*** [$CheckMac(7(e_2^s), nackkeyb1) = ok$] ***or***
[$CheckMac(7(e_3^s), nackkeyb1) = ok$] *checkNACKMACKIb2*
else $\overline{c_{badpck}}\langle$BadControl$\rangle$.*Int(*$e_{1-3}^i$*)*;

checkNACKMACKIb2 $\stackrel{def}{=}$
[$CheckMac(7(e_1^s), nackkeyb2) = ok$] ***or*** [$CheckMac(7(e_2^s), nackkeyb2) = ok$] ***or***
[$CheckMac(7(e_3^s), nackkeyb2) = ok$] *hndleACKNACKI2*(s, d, apID, e_{1-3}^i, sID, acknum, b1, b2)
else $\overline{c_{badpck}}\langle$BadControl$\rangle$.*Int(*$e_{1-3}^i$*)*;

checkNACKMACKIb1b2b3 $\stackrel{def}{=}$
[$CheckMac(7(e_1^s), nackkeyb1) = ok$] ***or*** [$CheckMac(7(e_2^s), nackkeyb1) = ok$] ***or***
[$CheckMac(7(e_3^s), nackkeyb1) = ok$] *checkNACKMACKIb2b3*
else $\overline{c_{badpck}}\langle$BadControl$\rangle$.*Int(*$e_{1-3}^i$*)*;

checkNACKMACKIb2b3 $\stackrel{def}{=}$
[$CheckMac(7(e_1^s), nackkeyb2) = ok$] ***or*** [$CheckMac(7(e_2^s), nackkeyb2) = ok$] ***or***
[$CheckMac(7(e_3^s), nackkeyb2) = ok$] *checkNACKMACKIb3*
else $\overline{c_{badpck}}\langle$BadControl$\rangle$.*Int(*$e_{1-3}^i$*)*;

checkNACKMACKIb3 $\stackrel{def}{=}$
[$CheckMac(7(e_1^s), nackkeyb3) = ok$] ***or*** [$CheckMac(7(e_2^s), nackkeyb3) = ok$] ***or***
[$CheckMac(7(e_3^s), nackkeyb3) = ok$] *hndleACKNACKI3*(s, d, apID, e_{1-3}^i, sID, acknum, b1, b2, b3)
else $\overline{c_{badpck}}\langle$BadControl$\rangle$.*Int(*$e_{1-3}^i$*)*;

For the case of the source node S, we extend the behavior of process *Src* specified in the real version with the subprocess

$$c_{privSI}(x^{rcv}_{should}).c_{si}(x^{rcv}_{really}).[x^{rcv}_{should} = x^{rcv}_{really}]\ procHndleMsgS;$$

where *procHndleMsgS* represents the process in which S continues to handle the received message (either a data packet or a control message) in the similar way as in the real version. More specifically, *procHndleMsgS* can be *fwdDt*(s, d, apID, e_{1-3}^i, sID, sq), *hndleACKS*(s, d, apID, e_{1-3}^i, sID, acknum), or *hndleACKNACKS1*(s, d, apID, e_{1-3}^i, sID, acknum, b1), *hndleACKNACKS2*(s, d, apID, e_{1-3}^i, sID, acknum, b1, b2), *hndleACKNACKS3*(s, d, apID, e_{1-3}^i, sID, acknum, b1, b2, b3).

Further, in a similar concept we have the addition output action of the constant *BadControl*, $\overline{c_{badpck}}\langle$BadControl$\rangle$, when the verification of ACK MAC or NACK MAC fails.

$rcvACKS(e^i_{1-3}) \stackrel{def}{=}$
c_{siACK}(acknum, ackkey).
$[CheckMac(6(e^s_1), ackkey) = ok]$ ***or*** $[CheckMac(6(e^s_2), ackkey) = ok]$ ***or***
$[CheckMac(6(e^s_3), ackkey) = ok]$ $hndleACK$(s, d, apID, e^s_{1-3}, sID, acknum) ***else***
$\overline{c_{badpck}}\langle$BadControl$\rangle$.$Src$(s, d, apID, e^s_{1-3}, sID, earAtmp);

the expected data on channel c_{siACK} is the extended by ackkey that represents per-packet ACK key. When S receives a NACK message it examines the ACK MAC and NACK MAC with the included keys, and it outputs the constant *BadControl* if the verification fails:

$rcvNACKS$(s, d, apID, e^i_{1-3}, sID, earAtmp)$\stackrel{def}{=}$
c_{siNACK}(acknum, b1, ackkey, nackkeyb1).$[CheckMac(6(e^s_1), ackkey) = ok]$ ***or***
$[CheckMac(6(e^s_2), ackkey) = ok]$ ***or*** $[CheckMac(6(e^s_3), ackkey) = ok]$
$checkNACKMACKSb1$ ***else*** $\overline{c_{badpck}}\langle$BadControl$\rangle$. Src(s, d, apID, e^s_{1-3}, sID, earAtmp);
[ ]
c_{siNACK}(acknum, b1, b2, ackkey, nackkeyb1, nackkeyb2).$[CheckMac(6(e^s_1), ackkey) = ok]$ ***or***
$[CheckMac(6(e^s_2), ackkey) = ok]$ ***or*** $[CheckMac(6(e^s_3), ackkey) = ok]$
$checkNACKMACKSb1b2$ ***else*** $\overline{c_{badpck}}\langle$BadControl$\rangle$. Src(s, d, apID, e^s_{1-3}, sID, earAtmp);
[ ]
c_{idNACK}(acknum,b1, b2, b3, ackkey, nackkeyb1, nackkeyb2, nackkeyb3).
$[CheckMac(6(e^s_1), ackkey) = ok]$ ***or*** $[CheckMac(6(e^s_2), ackkey) = ok]$ ***or***
$[CheckMac(6(e^s_3), ackkey) = ok]$
$checkNACKMACKSb1b2b3$ ***else*** $\overline{c_{badpck}}\langle$BadControl$\rangle$. Src(s, d, apID, e^s_{1-3}, sID, earAtmp);

$checkNACKMACKSb1 \stackrel{def}{=}$
$[CheckMac(7(e^s_1), nackkeyb1) = ok]$ ***or*** $[CheckMac(7(e^s_2), nackkeyb1) = ok]$ ***or***
$[CheckMac(7(e^s_3), nackkeyb1) = ok]$ $hndleACKNACKS1$(s, d, apID, e^i_{1-3}, sID, acknum, b1)
else $\overline{c_{badpck}}\langle$BadControl$\rangle$. Src(s, d, apID, e^s_{1-3}, sID, earAtmp);

$checkNACKMACKSb1b2 \stackrel{def}{=}$
$[CheckMac(7(e^s_1), nackkeyb1) = ok]$ ***or*** $[CheckMac(7(e^s_2), nackkeyb1) = ok]$ ***or***
$[CheckMac(7(e^s_3), nackkeyb1) = ok]$ $checkNACKMACKSb2$
else $\overline{c_{badpck}}\langle$BadControl$\rangle$. Src(s, d, apID, e^s_{1-3}, sID, earAtmp);

$checkNACKMACKSb2 \stackrel{def}{=}$
$[CheckMac(7(e^s_1), nackkeyb2) = ok]$ ***or*** $[CheckMac(7(e^s_2), nackkeyb2) = ok]$ ***or***
$[CheckMac(7(e^s_3), nackkeyb2) = ok]$ $hndleACKNACKS2$(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2)
else $\overline{c_{badpck}}\langle$BadControl$\rangle$. Src(s, d, apID, e^s_{1-3}, sID, earAtmp);

$checkNACKMACKSb1b2b3 \stackrel{def}{=}$
$[CheckMac(7(e^s_1), nackkeyb1) = ok]$ ***or*** $[CheckMac(7(e^s_2), nackkeyb1) = ok]$ ***or***
$[CheckMac(7(e^s_3), nackkeyb1) = ok]$ $checkNACKMACKSb2b3$
else $\overline{c_{badpck}}\langle$BadControl$\rangle$. Src(s, d, apID, e^s_{1-3}, sID, earAtmp);

$checkNACKMACKSb2b3 \stackrel{def}{=}$
$[CheckMac(7(e^s_1), nackkeyb2) = ok]$ ***or*** $[CheckMac(7(e^s_2), nackkeyb2) = ok]$ ***or***
$[CheckMac(7(e^s_3), nackkeyb2) = ok]$ $checkNACKMACKSb3$
else $\overline{c_{badpck}}\langle$BadControl$\rangle$. Src(s, d, apID, e^s_{1-3}, sID, earAtmp);

$checkNACKMACKSb3 \stackrel{def}{=}$
$[CheckMac(7(e^s_1), nackkeyb3) = ok]$ ***or*** $[CheckMac(7(e^s_2), nackkeyb3) = ok]$ ***or***
$[CheckMac(7(e^s_3), nackkeyb3) = ok]$ $hndleACKNACKS3$(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2, b3)
else $\overline{c_{badpck}}\langle$BadControl$\rangle$. Src(s, d, apID, e^s_{1-3}, sID, earAtmp);

Finally, for D the behavior of process *Dst* specified in the real version is extended with the subprocess

$$c_{privID}(x^{rcv}_{should}).c_{id}(x^{rcv}_{really}).[x^{rcv}_{should} = x^{rcv}_{really}]\ hndleDtDst;$$

where *hndleDtDst* represents the process in which D continues to handle the received data packets specified in the real version.

C The specification and verification of DTSN and SDTP in the PAT process analysis toolkit

C.1 Verifying the DTSN protocol

We specify the behavior of the attacker(s) in case of the DTSN protocol. We distinguish the following scenarios and examine the possible ability of the attacker(s). The behaviors of the attackers are defined as the processes *procA1()* and *procA2()*, which are composed of different subprocesses that we discuss below:

1. For the topology S - A1 - I - D
   - Without receiving any msg:
     - A1 sends a data packet to I

       ```
       PAT code:
       A1NotRcvSndPck2I()=
          chIAPck!seqA1.earA.rtxA -> procA1()
          [] chIAPck!seqA2.earA.rtxA -> procA1()
          (where earA = {0,1}; rtxA = {0,1})
       ```

       Due to the PAT process analysis toolkit is not optimized for handling security protocols in the presence of attackers with unlimited ability, to avoid running out of memory we assume that an attacker has no memory, namely, it can construct messages only based on the latest information it receives, or its owned data. In our model, by default the attackers have two sequence number *seqA1* and *seqA2* which are the smallest (i.e., 1) and the largest possible sequence number, respectively. The attackers, in addition, posses the pre-defined values *bA1*,..., *bA4* for requiring re-transmission in NACK messages. The code part *chIAPck!seqA1.ear.rtx -> procA1()* says that after sending the data packet *seqA1.ear.rtx*, A1 repeats its operation. We denote this sub-process of *procA1()* by *A1NotRcvSndPck2I()*.
     - A1 sends an ACK to I or S

       ```
       PAT code:
       A1NotRcvSndAck2I() =
         chIAAck!seqA1 -> procA1()
         [] chIAAck!seqA2 -> procA1()

       A1NotRcvSndAck2S() =
         chSAAck!seqA1 -> procA1()
         [] chSAAck!seqA2 -> procA1()
       ```

       The attacker constructs two ACK messages based on its knowledge and chooses to send one of them to node I or S. Note that we do not consider the scenario where A1 sends ACK to both nodes at the same time, because I will forward to S the ACK, which would lead to the same effect. We denote these sub-processes of *procA1()* by *A1NotRcvSndAck2I*() and *A1NotRcvSndAck2S*().
     - A1 sends a NACK to I or S

```
PAT code:
to I:
    A1NotRcvSndNack2I()=
      chIANack!seqA1.bA1 -> procA1()
      [] chIANack!seqA1.bA1.bA2 -> procA1()
      [] chIANack!seqA1.bA1.bA2.bA3 -> procA1()
      [] chIANack!seqA1.bA1.bA2.bA3.bA4 -> procA1()
      [] chIANack!seqA2.bA1 -> procA1()
      [] chIANack!seqA2.bA1.bA2 -> procA1()
      [] chIANack!seqA2.bA1.bA2.bA3 -> procA1()
      [] chIANack!seqA2.bA1.bA2.bA3.bA4 -> procA1()

to S:
    A1NotRcvSndNack2S()=
      chSANack!seqA1.bA1 -> procA1()
      [] chSANack!seqA1.bA1.bA2 -> procA1()
      [] chSANack!seqA1.bA1.bA2.bA3 -> procA1()
      [] chSANack!seqA1.bA1.bA2.bA3.bA4 -> procA1()
      [] chSANack!seqA2.bA1 -> procA1()
      [] chSANack!seqA2.bA1.bA2 -> procA1()
      [] chSANack!seqA2.bA1.bA2.bA3 -> procA1()
      [] chSANack!seqA2.bA1.bA2.bA3.bA4 -> procA1()
```

The attacker A1 sends the NACKs to I or S using the elements it posseses. We denote these sub-processes of *procA1()* by *A1NotRcvSndNack2I*() and *A1NotRcvSndNack2S*().

- After receiving a data pck, *chSAPck?seq.ear.rtx*:
  - A1 sends a data packet to I

```
A1RcvPckSndPck2I()=
  chIAPck!seqA1.ear.rtx -> procA1()
  [] chIAPck!seqA2.ear.rtx -> procA1()
  [] chIAPck!seq.near.rtx -> procA1()
  [] chIAPck!seq.ear.nrtx -> procA1()
  [] chIAPck!seq.near.nrtx -> procA1()
  [] chIAPck!seqA1.near.rtx -> procA1()
  [] chIAPck!seqA1.ear.nrtx -> procA1()
  [] chIAPck!seqA1.near.nrtx -> procA1()
  [] chIAPck!seqA2.near.rtx -> procA1()
  [] chIAPck!seqA2.ear.nrtx -> procA1()
  [] chIAPck!seqA2.near.nrtx -> procA1()
```

In this case, besides the sequence numbers *seqA1* and *seqA2* the attacker can use the received *seq*, *ear* and *rtx*. The notations *near* and *nrtx* represent the negation of *ear* and *rtx*. We denote this sub-processes by *A1RcvPckSndPck2I*().

  - A1 sends an ACK to I or S

```
A1RcvPckSndAck2I()=
  chIAAck!seqA1 -> procA1()
  [] chIAAck!seqA2 -> procA1()
  [] chIAAck!seq -> procA1()

A1RcvPckSndAck2S()=
  chSAAck!seqA1 -> procA1()
  [] chSAAck!seqA2 -> procA1()
  [] chSAAck!seq -> procA1()
```

  - A1 sends a NACK to I or S

```
to I:

A1RcvPckSndNack2I()=
 A:{seqA1,seqA2,seq};B1:{bA1,seq}@chIANack!A.B1 -> procA1()
 [] A:{seqA1,seqA2,seq};B1:{bA1,seq};
    B2:{bA2,seq}@chIANack!A.B1.B2 -> procA1()
 [] A:{seqA1,seqA2,seq};B1:{bA1,seq};B2:{bA2,seq};
    B3:{bA3,seq}@chIANack!A.B1.B2.B3 -> procA1()
 [] A:{seqA1,seqA2,seq};B1:{bA1,seq};B2:{bA2,seq};
    B3:{bA3,seq};B4:{bA4,seq}@chIANack!A.B1.B2.B3.B4 -> procA1()

to S:

A1RcvPckSndNack2S()=
 A:{seqA1,seqA2,seq};B1:{bA1,seq}@chSANack!A.B1 -> procA1()
 [] A:{seqA1,seqA2,seq};B1:{bA1,seq};
    B2:{bA2,seq}@chSANack!A.B1.B2 -> procA1()
 [] A:{seqA1,seqA2,seq};B1:{bA1,seq};B2:{bA2,seq};
    B3:{bA3,seq}@chSANack!A.B1.B2.B3 -> procA1()
 [] A:{seqA1,seqA2,seq};B1:{bA1,seq};B2:{bA2,seq};
    B3:{bA3,seq};B4:{bA4,seq}@chSANack!A.B1.B2.B3.B4 -> procA1()
```

The attacker A1 chooses an option to send NACK message which is composed of the information available to the attacker. Beside the pre-defined knowledge, the attacker can use the received *seq* as well. The language constuct

x:{v1..vn}; y:w1..wm@Proc(x,y)

represents the operation of the process *Proc* with the variables x and y whose values are taken from the sets *v1..vn* and w1..wm, respectively.

- After receiving an ACK, *chIAAck?ack*:
  - A1 sends a data packet to I

```
A1RcvAckSndPck2()=
      chIAPck!seqA1.ear.rtx -> procA1()
      [] chIAPck!seqA2.ear.rtx -> procA1()
      [] chIAPck!ack.ear.rtx -> procA1()
```

  - A1 sends an ACK to I or S

```
A1RcvAckSndAck2I()=
      chIAAck!seqA1 -> procA1()
      [] chIAAck!seqA2 -> procA1()
      [] chIAAck!ack -> procA1()

A1RcvAckSndAck2S()=
      chSAAck!seqA1 -> procA1()
      [] chSAAck!seqA2 -> procA1()
      [] chSAAck!ack -> procA1()
```

  - A1 sends a NACK to I or S

```
to I:
A1RcvAckSndNack2I()=
 A:{seqA1,seqA2,ack};B1:{bA1,ack}@chIANack!A.B1 -> procA1()
 [] A:{seqA1,seqA2,ack};B1:{bA1,ack};
    B2{bA2,ack}@chIANack!A.B1.B2 -> procA1()
 [] A:{seqA1,seqA2,ack};B1:{bA1,ack};B2:{bA2,ack};
    B3:{bA3,ack}@chIANack!A.B1.B2.B3 -> procA1()
```

```
 [] A:{seqA1,seqA2,ack};B1:{bA1,ack};B2:{bA2,ack};
    B3:{bA3,ack}; B4:{bA4,ack}@chIANack!A.B1.B2.B3.B4 -> procA1()

A1RcvAckSndNack2S()=
to S:
 A:{seqA1,seqA2,ack};B1:{bA1,ack}@chSANack!A.B1 -> procA1()
 [] A:{seqA1,seqA2,ack};B1:{bA1,ack};
    B2{bA2,ack}@chSANack!A.B1.B2 -> procA1()
 [] A:{seqA1,seqA2,ack};B1:{bA1,ack};B2:{bA2,ack};
    B3:{bA3,ack}@chSANack!A.B1.B2.B3 -> procA1()
 [] A:{seqA1,seqA2,ack};B1:{bA1,ack};B2:{bA2,ack};
    B3:{bA3,ack};B4:{bA4,ack}@chSANack!A.B1.B2.B3.B4 -> procA1()
```

- After receiving a NACK *chIANack?ack.b1* [] *chIANack?ack.b1.b2* [] *chIANack?ack.b1.b2.b3* [] *chIANack?ack.b1.b2.b3.b4*:
  - A1 sends a data packet to I

```
 A1RcvNack-1Bit-SndPck2I()=
    chIAPck!seqA1.earA.rtxA -> procA1()
    [] chIAPck!seqA2.earA.rtxA -> procA1()
    [] chIAPck!ack.earA.rtxA -> procA1()
    [] chIAPck!b1.earA.rtxA -> procA1()

A1RcvNack-2Bit-SndPck2I()=
    chIAPck!seqA1.earA.rtxA -> procA1()
    [] chIAPck!seqA2.earA.rtxA -> procA1()
    [] chIAPck!ack.earA.rtxA -> procA1()
    [] chIAPck!b1.earA.rtxA -> procA1()
    [] chIAPck!b2.earA.rtxA -> procA1()

A1RcvNack-3Bit-SndPck2I()=
    chIAPck!seqA1.earA.rtxA -> procA1()
    [] chIAPck!seqA2.earA.rtxA -> procA1()
    [] chIAPck!ack.earA.rtxA -> procA1()
    [] chIAPck!b1.earA.rtxA -> procA1()
    [] chIAPck!b2.earA.rtxA -> procA1()
    [] chIAPck!b3.earA.rtxA -> procA1()

A1RcvNack-4Bit-SndPck2I()=
    chIAPck!seqA1.ear.rtx -> procA1()
    [] chIAPck!seqA2.earA.rtxA -> procA1()
    [] chIAPck!ack.earA.rtxA -> procA1()
    [] chIAPck!b1.earA.rtxA -> procA1()
    (earA = {0,1}; rtxA = {0,1}) & we assume that b1, ..., b4 are seqnum
    of the packets to be RTXd.
```

  - A1 sends an ACK to I or S

```
to I:
A1RcvNack-1Bit-SndAck2I()=
    chIAAck!seqA1 -> procA1()
    [] chIAAck!seqA2 -> procA1()
    [] chIAAck!ack -> procA1()
    [] chIAAck!b1 -> procA1()

A1RcvNack-2Bit-SndAck2I()=
    chIAAck!seqA1 -> procA1()
```

```
    [] chIAAck!seqA2 -> procA1()
    [] chIAAck!ack -> procA1()
    [] chIAAck!b1 -> procA1()
    [] chIAAck!b2 -> procA1()

A1RcvNack-3Bit-SndAck2I()=
    chIAAck!seqA1 -> procA1()
    [] chIAAck!seqA2 -> procA1()
    [] chIAAck!ack -> procA1()
    [] chIAAck!b1 -> procA1()
    [] chIAAck!b2 -> procA1()
    [] chIAAck!b3 -> procA1()

A1RcvNack-4Bit-SndAck2I()=
    chIAAck!seqA1 -> procA1()
    [] chIAAck!seqA2 -> procA1()
    [] chIAAck!ack -> procA1()
    [] chIAAck!b1 -> procA1()
    [] chIAAck!b2 -> procA1()
    [] chIAAck!b3 -> procA1()
    [] chIAAck!b4 -> procA1()


to S:
A1RcvNack-1Bit-SndAck2S()=
    chSAAck!seqA1 -> procA1()
    [] chSAAck!seqA2 -> procA1()
    [] chSAAck!ack -> procA1()
    [] chSAAck!b1 -> procA1()

A1RcvNack-2Bit-SndAck2S()=
    chSAAck!seqA1 -> procA1()
    [] chSAAck!seqA2 -> procA1()
    [] chSAAck!ack -> procA1()
    [] chSAAck!b1 -> procA1()
    [] chSAAck!b2 -> procA1()

A1RcvNack-3Bit-SndAck2S()=
    chSAAck!seqA1 -> procA1()
    [] chSAAck!seqA2 -> procA1()
    [] chSAAck!ack -> procA1()
    [] chSAAck!b1 -> procA1()
    [] chSAAck!b2 -> procA1()
    [] chSAAck!b3 -> procA1()

A1RcvNack-4Bit-SndAck2S()=
    chSAAck!seqA1 -> procA1()
    [] chSAAck!seqA2 -> procA1()
    [] chSAAck!ack -> procA1()
    [] chSAAck!b1 -> procA1()
    [] chSAAck!b2 -> procA1()
    [] chSAAck!b3 -> procA1()
    [] chSAAck!b4 -> procA1()
```

– A1 sends a NACK to I or S

```
to I:
 snd NACK to I, after the action chIANack?ack.b1:

A1RcvNack-1Bit-SndNack2I-1Bit()=
 A:{seqA1,seqA2,ack,b1};B1:{bA1,ack,b1}@chIANack!A.B1 -> procA1()
 [] A:{seqA1,seqA2,ack,b1};B1:{bA1, ack,b1};B2:{bA2,ack,b1}
    @chIANack!A.B1.B2 -> procA1()
 [] A:{seqA1,seqA2,ack,b1};B1:{bA1,ack,b1};B2:{bA2,ack,b1};
    B3:{bA3,ack,b1}@chIANack!A.B1.B2.B3 -> procA1()
 [] A:{seqA1,seqA2,ack,b1}; B1:{bA1,ack,b1};B2:{bA2,ack,b1};
    B3:{bA3,ack,b1};B4:{bA4,ack,b1}@chIANack!A.B1.B2.B3.B4 -> procA1()


 snd NACK to I, after the action chIANack?ack.b1.b2:

A1RcvNack-2Bit-SndNack2I-2Bit()=
 A:{seqA1,seqA2,ack,b1,b2};B1:{bA1,ack,b1,b2}@chIANack!A.B1 -> procA1()
 [] A:{seqA1,seqA2,ack,b1,b2};B1:{bA1,ack,b1,b2};
    B2:{bA2,ack,b1,b2}@chIANack!A.B1.B2 -> procA1()
 [] A:{seqA1,seqA2,ack,b1,b2};B1:{bA1,ack,b1,b2};B2:{bA2,ack,b1,b2};
    B3:{bA3,ack,b1,b2}@chIANack!A.B1.B2.B3  -> procA1()
 [] A:{seqA1,seqA2,ack,b1,b2};B1:{bA1,ack,b1,b2};B2:{bA2,ack,b1,b2};
    B3:{bA3,ack,b1,b2};B4:{bA4,ack,b1,b2}
    @chIANack!A.B1.B2.B3.B4 -> procA1()

 snd NACK to I, after the actionchIANack?ack.b1.b2.b3:

A1RcvNack-3Bit-SndNack2I-3Bit()=
 A:{seqA1,seqA2,ack,b1,b2,b3};B1:{bA1,ack,b1,b2,b3}
 @chIANack!A.B1 -> procA1()
 [] A:{seqA1,seqA2,ack,b1,b2,b3};B1:{bA1,ack,b1,b2,b3};
    B2:{bA2,ack,b1,b2,b3}@chIANack!A.B1.B2 -> procA1()
 [] A:{seqA1,seqA2,ack,b1,b2,b3};B1:{bA1,ack,b1,b2,b3};B2:{bA2,ack,
    b1,b2,b3};B3:{bA3,ack,b1,b2,b3}@chIANack!A.B1.B2.B3 -> procA1()
 [] A:{seqA1,seqA2,ack,b1,b2,b3};B1:{bA1,ack,b1,b2,b3};
    B2:{bA2,ack,b1,b2,b3};B3:{bA3,ack,b1,b2,b3};
    B4:{bA4,ack,b1,b2,b3}@chIANack!A.B1.B2.B3.B4 -> procA1()

 snd NACK to I, after the action chIANack?ack.b1.b2.b3.b4:

A1RcvNack-4Bit-SndNack2I-4Bit()=
 A:{seqA1,seqA2,ack,b1,b2,b3,b4};B1:{bA1,ack,b1,b2,b3,b4}
 @chIANack!A.B1 -> procA1()
 [] A:{seqA1,seqA2, ack,b1,b2,b3,b4};B1:{bA1, ack,b1,b2,b3,b4}.
    B2:{bA2, ack,b1,b2,b3,b4}@chIANack!A.B1.B2 -> procA1()
 [] A:{seqA1,seqA2,ack,b1,b2,b3,b4};B2:{bA1,ack,b1,b2,b3,b4};
    B3:{bA2,ack,b1,b2,b3,b4};B4:{bA3,ack,b1,b2,b3,b4}
    @chIANack!A.B1.B2.B3 -> procA1()
 [] A:{seqA1,seqA2,ack,b1,b2,b3,b4};B1:{bA1,ack,b1,b2,b3,b4};
    B2:{bA2,ack,b1,b2,b3,b4};B3:{bA3,ack,b1,b2,b3,b4};
    B4:{bA4,ack,b1,b2,b3,b4}@chIANack!A.B1.B2.B3.B4 -> procA1()
```

```
to S:
 snd NACK to S, after the action chSANack?ack.b1:

A1RcvNack-1Bit-SndNack2S-1Bit()=
 A:{seqA1,seqA2,ack,b1};B1:{bA1,ack,b1}@chSANack!A.B1 -> procA1()
 [] A:{seqA1,seqA2,ack,b1};B1:{bA1, ack,b1};B2:{bA2,ack,b1}
    @chSANack!A.B1.B2 -> procA1()
 [] A:{seqA1,seqA2,ack,b1};B1:{bA1,ack,b1};B2:{bA2,ack,b1};
    B3:{bA3,ack,b1}@chSANack!A.B1.B2.B3 -> procA1()
 [] A:{seqA1,seqA2,ack,b1}; B1:{bA1,ack,b1};B2:{bA2,ack,b1};
    B3:{bA3,ack,b1};B4:{bA4,ack,b1}
    @chSANack!A.B1.B2.B3.B4 -> procA1()

A1RcvNack-2Bit-SndNack2S-2Bit()=
 A:{seqA1,seqA2,ack,b1,b2};B1:{bA1,ack,b1,b2}
 @chSANack!A.B1 -> procA1()
 [] A:{seqA1,seqA2,ack,b1,b2};B1:{bA1,ack,b1,b2};
    B2:{bA2,ack,b1,b2}@chSANack!A.B1.B2 -> procA1()
 [] A:{seqA1,seqA2,ack,b1,b2};B1:{bA1,ack,b1,b2};B2:{bA2,ack,b1,b2};
    B3:{bA3,ack,b1,b2}@chSANack!A.B1.B2.B3 -> procA1()
 [] A:{seqA1,seqA2,ack,b1,b2};B1:{bA1,ack,b1,b2};B2:{bA2,ack,b1,b2};
    B3:{bA3,ack,b1,b2};B4:{bA4,ack,b1,b2}
    @chSANack!A.B1.B2.B3.B4 -> procA1()

 snd NACK to S, after the action chSANack?ack.b1.b2.b3

A1RcvNack-3Bit-SndNack2S-3Bit()=
 A:{seqA1,seqA2,ack,b1,b2,b3};B1:{bA1,ack,b1,b2,b3}
 @chSANack!A.B1 -> procA1()
 [] A:{seqA1,seqA2,ack,b1,b2,b3};B1:{bA1,ack,b1,b2,b3};
    B2:{bA2,ack,b1,b2,b3}@chSANack!A.B1.B2 -> procA1()
 [] A:{seqA1,seqA2, ack,b1,b2,b3};B1:{bA1, ack,b1,b2,b3};
    B2:{bA2, ack,b1,b2,b3};B3:{bA3,ack,b1,b2,b3}
    @chSANack!A.B1.B2.B3 -> procA1()
 [] A:{seqA1,seqA2,ack,b1,b2,b3};B1:{bA1,ack,b1,b2,b3};
    B2:{bA2,ack,b1,b2,b3};B3:{bA3,ack,b1,b2,b3};
    B4:{bA4,ack,b1,b2,b3}@chSANack!A.B1.B2.B3.B4 -> procA1()

 snd NACK to S, after the action chSANack?ack.b1.b2.b3.b4:

A1RcvNack-4Bit-SndNack2S-4Bit()=
 A:{seqA1,seqA2,ack,b1,b2,b3,b4};B1:{bA1,ack,b1,b2,b3,b4}
 @chSANack!A.B1 -> procA1()
 [] A:{seqA1,seqA2, ack,b1,b2,b3,b4};B1:{bA1, ack,b1,b2,b3,b4}.
    B2:{bA2,ack,b1,b2,b3,b4}@chSANack!A.B1.B2 -> procA1()
 [] A:{seqA1,seqA2,ack,b1,b2,b3,b4};B2:{bA1,ack,b1,b2,b3,b4};
    B3:{bA2,ack,b1,b2,b3,b4};B4:{bA3,ack,b1,b2,b3,b4}
    @chSANack!A.B1.B2.B3 -> procA1()
 [] A:{seqA1,seqA2,ack,b1,b2,b3,b4};B1:{bA1,ack,b1,b2,b3,b4};
    B2:{bA2,ack,b1,b2,b3,b4};B3:{bA3,ack,b1,b2,b3,b4};
    B4:{bA4,ack,b1,b2,b3,b4}@chSANack!A.B1.B2.B3.B4 -> procA1()
```

The behavior of the attacker A1 is specified by process procA1() which is composed of the non-deterministic choices of each sub-process.

```
ProcA1() =
 A1NotRcvSndPck2I()
 [] A1NotRcvSndAck2I() [] A1NotRcvSndNack2I()
 [] A1NotRcvSndAck2S() [] A1NotRcvSndNack2S()
 [] chSAPck?seq.ear.rtx ->
    (
     A1RcvPckSndPck2I() [] A1RcvPckSndAck2S()
     [] A1RcvPckSndNack2I() [] A1RcvPckSndNack2S()
     [] chIAPck!seq.ear.rtx -> procA1()
    )
 [] chIAPck?ack ->
    (
     A1RcvAckSndPck2I()
     [] A1RcvAckSndAck2I() [] A1RcvAckSndAck2S()
     [] A1RcvAckSndNack2I() [] A1RcvAckSndNack2S()
     [] chSAAck!ack -> procA1()
    )
 [] chIAPck?ack.b1 ->
    (
     A1RcvNack-1Bit-SndPck2I() []
     [] A1RcvNack-1Bit-SndAck2I() [] A1RcvNack-1Bit-SndAck2S()
     [] A1RcvNack-1Bit-SndNack2I-1Bit() [] A1RcvNack-1Bit-SndNack2S-1Bit()
     [] chSAAck!ack.b1 -> procA1()
    )
 [] chIAPck?ack.b1.b2 ->
    (
     A1RcvNack-2Bit-SndPck2I() []
     [] A1RcvNack-2Bit-SndAck2I() [] A1RcvNack-2Bit-SndAck2S()
     [] A1RcvNack-2Bit-SndNack2I-1Bit() [] A1RcvNack-2Bit-SndNack2I-2Bit()
     [] A1RcvNack-2Bit-SndNack2S-1Bit() [] A1RcvNack-2Bit-SndNack2S-2Bit()
     [] chSAAck!ack.b1.b2 -> procA1()
    )
 [] chIAPck?ack.b1.b2.b3 ->
    (
     A1RcvNack-3Bit-SndPck2I() []
     [] A1RcvNack-3Bit-SndAck2I() [] A1RcvNack-4Bit-SndAck2S()
     [] A1RcvNack-3Bit-SndNack2I-1Bit() [] A1RcvNack-3Bit-SndNack2I-2Bit()
     [] A1RcvNack-3Bit-SndNack2I-3Bit()
     [] A1RcvNack-3Bit-SndNack2S-1Bit() [] A1RcvNack-3Bit-SndNack2S-2Bit()
     [] A1RcvNack-3Bit-SndNack2S-3Bit()
     [] chSAAck!ack.b1.b2.b3 -> procA1()
    )
 [] chIAPck?ack.b1.b2.b3.b4 ->
    (
     A1RcvNack-4Bit-SndPck2I() []
     [] A1RcvNack-4Bit-SndAck2I() [] A1RcvNack-4Bit-SndAck2S()
     [] A1RcvNack-4Bit-SndNack2I-1Bit() [] A1RcvNack-4Bit-SndNack2I-2Bit()
     [] A1RcvNack-4Bit-SndNack2I-3Bit() [] A1RcvNack-4Bit-SndNack2I-4Bit()
     [] A1RcvNack-4Bit-SndNack2S-1Bit() [] A1RcvNack-4Bit-SndNack2S-2Bit()
     [] A1RcvNack-4Bit-SndNack2S-3Bit() [] A1RcvNack-4Bit-SndNack2S-4Bit()
     [] chSAAck!ack.b1.b2.b3.b4 -> procA1()
    )
```

The last choice at each case/branch represents the scenario when the attacker follows the protocol and forwards the correct messages to I and S.

The DTSN protocol with the second topology is specified as the parallel compositions of each honest node and the attacker A1:

$$\text{DTSNA1}() = \text{procS}() \;|||\; \text{procA1}() \;|||\; \text{procI}() \;|||\; \text{procD}()$$

2. For the second topology S - I - A2 - D, the scenarios and PAT codes are the same as in the case of the first topology S - A1 - I - D except that the used channels at each corresponding step are changed as follows: In the first topology, A1 receives data packets from S on *chSAPck*, which is changed to *chIAPck* in the second case because now data packets come from I. Similarly, the inputs on *chSAAck* and *chSANack* are changed to *chDAAack* and *chDANack*, respectively. The outputs by A on *chIAPck*, *chIAAck*, *chIANack*, *chSAAck* and *chSANack* are changed to *chDAPck*, *chDAAck*, *chDANack*, *chIAAck* and *chIANack*, respectively. The attacker process *procA2()* describing the behavior of A2 is specified in the same way as *procA1()*, but with different channels.

   The DTSN protocol with the second topology is specified as the following parallel compositions:

$$\text{DTSNA2}() = \text{UpLayer}() \;|||\; \text{procS}() \;|||\; \text{procI}() \;|||\; \text{procA2}() \;|||\; \text{procD}()$$

3. For the third topology S - A1 - I - A2 - D, we apply both the specification of processes A1 and A2. The DTSN protocol with the third topology is specified as the parallel compositions of each honest node and the two attackers:

$$\text{DTSNA1A2}() = \text{UpLayer}() \;|||\; \text{procS}() \;|||\; \text{procA1}() \;|||\; \text{procI}() \;|||\; \text{procA2}() \;|||\; \text{procD}()$$

C.2 Verifying the SDTP protocol

For the SDTP protocol, there are the same three topologies and the attacker's behavior scenarios are specified similarly as in case of DTSN, however, with different message formats. In the following, we only discuss the first some scenarios in detail, the rest parts are based on the similar concepts and can be continued in a straightforward way:

1. For the topology S - A1 - I - D
   - Without receiving any msg (SDTP): (no memory)
     - A1 sends a data packet to I

```
PAT code:
chIAPck!seqA1.near.nrtx.seqA1.seqA1.Katt.seqA1.seqA1.Katt
[] chIAPck!seqA2.seqA2.earA.rtxA.seqA2.seqA2.Katt.seqA2.seqA2.Katt
[] chIAPck!seqA2.seqA2.earA.rtxA.seqA1.seqA1.Katt.seqA1.seqA1.Katt
[] chIAPck!seqA1.seqA1.earA.rtxA.seqA2.seqA2.Katt.seqA2.seqA2.Katt
[] chIAPck!seqA2.seqA2.earA.rtxA.seqA1.seqA1.Katt.seqA1.seqA1.Katt
[] chIAPck!seqA1.seqA1.earA.rtxA.seqA2.seqA2.Katt.seqA2.seqA2.Katt
(where earA = {0,1}; rtxA = {0,1})
```

       The attacker, according to the data packet's format in SDTP, includes combinations of the ACK and NACK MACs in each sent message. Recall that the ACK MAC is represented in the form *seq.seq.Kack*, where the first *seq* represents the data packet with the sequence number *seq*, and the second part *seq.Kack* represents the the ACK key corresponding to *seq*. Hence, the triple *seq.seq.Kack* is the ACK MAC computed on the packet *seq* using the ACK key *seq.Kack*. Similarly, *seq.seq.Knack* represents the NACK MAC of the packet *seq*. The attackers does not posses the two

master keys *Kack* and *Knack* of honest nodes, but only their keys *Katt*. Because honest nodes are specified to waiting for these MACs format, the attackers should compose the MACs in this format as well, namely, *seqA.seqA.Katt*.

- On receiving a data on chSAPck, *chSAPck?seq.ear.rtx.seq1.seq2.Kack.seq3.seq4.Knack*:
  - A1 sends a data packet to I:
    Regarding the options in the choice defined for DTSN, A1 includes the corresponding ACK MAC and NACK MACs in each sent data packets. In addition, these choice options are extended with the branches with the data packets including the combinations of available ACK/NACK MACs *seq1.seq2.Kack*, *seq3.seq4.Knack*, *seqA1.seqA1.Katt*, *seqA2.seqA2.Katt*, *seqA2.seqA2.Katt*, and *seqA2.seqA2.Katt*.
  - A1 sends an ACK to I or S:
    Similarly, A1 extends the sent ACK messages defined in DTSN with a combinations of the incorrect ACK and NACK keys, *seqA1.Katt* and *seqA2.Katt*.
  - A1 sends a NACK to I or S: For each of the four cases, A1 extends the sent NACK messages defined in DTSN with a combinations of the incorrect ACK and NACK keys, *seqA1.Katt* and *seqA2.Katt*.

The rest cases are defined in a very similar way, by extending the scenarios and the PAT codes given in DTSN with different combinations of ACK/NACK MACs, and ACK/NACK keys.

In the following, we specify the bad states for DTSN and SDTP, and run the model-checking if these bad states can be reached. The bad states, and the verification goals can be defined in PAT's language in the form of logical formulae and assertions, respectively.

Let the number of buffer entries that are freed at node I after receiving an ACK/NACK message, be *freenum*, and the number of packets received in sequence by node D be *acknum*. The bad state *BadState1* specifies the state where (*freenum* > *acknum*).

```
PAT code:
   #define BadState1 (freenum > acknum)
   #assert SDTPA1A2() reaches BadState1
   #assert SDTPsubA1subA2() reaches BadState1
```

In case the process *SDTPA1A2()* reaches *BadState*, it can be seen as a security hole or an undesired property of SDTP, because according to the definition of I, it should always empty at most as many entries as the ack number it receives in ACK/NACK messages, which is *acknum*. Note that we can also define a bad state as (*freenum* > *rcvnum*), where *rcvnum* is the number of received packets (not necessarily in sequence).

Let us denote *SDTPsubA1subA2()* as the process which, intead of the whole processes, includes different sub-processes of *procA1()* and *procA2()*. To shorten the code and reduce the complexity of the verification, besides considering *SDTPA1A2()*, we also examine if there is any *SDTPsubA1subA2()* that reaches *BadState1*. In case there is such a *SDTPsubA1subA2()* then it follows that the "bigger" *SDTPA1A2()* reaches *BadState1* as well.

After running the PAT model-checker with

```
subA1() =
  A1NotRcvSndPck2I()
  [] chSAPck?seq.ear.rtx ->
     (
       A1RcvPckSndPck2I() [] chIAPck!seq.ear.rtx -> subA1()
     )
subA2() =
  A1NotRcvSndAck2I()
  [] chIAPck?seq.ear.rtx -> A1RcvPckSndAck2I()
```

```
SDTPsubA1subA2() =
  UpLayer() ||| procS() ||| subA1() ||| procI() ||| subA2() ||| procD()
```

the tool returned *Valid* for the second assertion along with the following scenario:

```
1. A1 sends I a data pck seqA2.ear.rtx: chIAPck!seqA2.ear.rtx
2. I stores this pck and forwards it to A2: chIAPck!seqA2.ear.rtx
3. A2, after the input chIAPck?seqA2.ear.rtx, sends to I seqA2: chIAAck!seqA2
4. As result, I deletes all the packets stored in its buffer.
```

As the result, basically, the attackers A1 and A2 can always achieve that the buffer of I is emptied, because by definition seqA2 is the largest possible sequence number, hence will be larger than every seq number stored by I. In the worst case, node I is always prevented from caching packets which corrupts the design goal of SDTP. For the first assertion, because of the complexity of the attacker processes and the search algorithm used by PAT, the model-checker returns *Valid* after a bigger amount of time and with a much longer attack trace. This long trace essentially shows the same type of attack as the shorter one, detected in the first assertion.

Note that in case of SDTP, PAT returns an attack scenario where both S and I empty their cache, with the unchanged *subA2()* and the subprocesses *subA1()*

```
subA1() =
  A1NotRcvSndPck2I()
  [] chSAPck?seq.ear.rtx ->
     (
       A1RcvPckSndPck2I() [] chIAPck!seq.ear.rtx -> subA1()
     )
  [] chIAAck?ack ->
     (
       A1RcvAckSndAck2S() [] chIAPck!ack -> subA1()
     )
```

in which A1 forwards the ACK message it received from I to node S.

In the following we examine the existence of this vulnerability for the case of SDTP. We take the same *BadState1* and consider the similar assertions.

```
#assert SDTPA1A2() reaches BadState1
#assert SDTPsubA1subA2() reaches BadState1
```

SDTPA1A2() and *SDTPsubA1subA2()* are defined in the similar concept as in DTSN. After running the PAT model-checker with

```
subA1() =
 A1NotRcvSndPck2I()
 [] chSAPck?seq.ear.rtx.seq1.seq2.Kack.seq3.seq4.Knack ->
   (
    A1RcvPckSndPck2I()
    [] chIAPck!seq.ear.rtx.seq1.seq2.Kack.seq3.seq4.Knack -> subA1()
   )

subA2() =
 A1NotRcvSndAck2I()
 [] chIAPck?seq.ear.rtx.seq1.seq2.Kack.seq3.seq4.Knack -> A1RcvPckSndAck2I()


SDTPsubA1subA2() =
  UpLayer() ||| procS() ||| subA1() ||| procI() ||| subA2() ||| procD()
```

the tool returned *Valid* for the second assertion along with the following scenario:

```
1. A1 sends to I a data pck seqA2.ear.rtx with the corresponding
   ACK MACs: seqA2.seqA2.Katt and NACK MACs: seqA2.seqA2.Katt
2. I stores this pck and forwards it (unchanged) to A2.
3. A2 received this packet and sends to I the ACK for seqA2:
   seqA2.seqA2.Katt.seqA2.Katt with the 2 keys seqA2.Katt and seqA2.Katt
4. As result, I deletes all the packets stored in its buffer because
   the key seqA2.Katt and the MACK seqA2.seqA2.Katt matches.
```

In summary, we get the result that both DTSN and SDTP are susceptible for this sandwich style attack scenario.

C.3 Some additional vulnerabilities of DTSN and SDTP found with PAT

To examine the next potential vulnerability of DTSN and SDTP, we specify the bad state in which the buffer of node I is not empty, however, I does not re-transmit any message during the session. In effect, the attackers can prevent DTSN and SDTP from achieving their design goal.

For the DTSN protocol, let us consider the following attack scenario: S sends data packets to A1, each case A1 replaces (with probability 1) the correct sequence number in them to non-existent numbers and sends these modified packets to I. Node I stores some of these packets and forwards them to A2, who modified each packet back to the correct ones and sends it to D (we assume that there is a side channel between node A1 and A2, hence, they can share information with each other. This is modelled by using the same channels *chIA*, *chSA* for A1 and A2. Note that in case we do not want any side channel between attackers we should use different channels between A1, A2 and I, S for example *chIA1*, *chSA1*, *chIA2*, *chSA2*).

This can be seen as an attack scenario because node I will never retransmit, since it stores non-existent sequence numbers, and according to the definition of DTSN, re-transmission only happens when node I stores the particular packet. This means that the attackers can corrupt the design goal of DTSN and bringing it back to the end-to-end retransmission scheme, which is an undesired status.

The verification technique for this attack scenario using PAT is as follows:

1. In the first step, we show the (maximum) probability that node I stores at least one packet sent by A1 is larger than 0 (or some percent). We run PAT for the assertion *DTSNA1A2() reaches goal with pmax*, where *goal* is ($BufIL \geq 1$) with *BufIL* representing the number of occupied entries in the buffer.

2. In the second step

   - We define *DTSNA1A2InotRTX()*, such version of DTSN in which node I never re-transmit. The process is the same as the original *DTSNA1A2()* except that when receiving ACK/NACK messages I does not re-transmit to D. Specifically, only the output actions of the form *IDpck!seq.ear.1* are removed from *DTSNA1A2()*.
   - We define *DTSNA1A2Ircvdwith1()* in which we set the probability that I receives a packet sent by A1 to 1. Namely, we concentrate on the case when node I received at least one packet fron A1, which can happen as it is already proved in the first point.
   - We run PAT for the assertion *DTSNA1A2Ircvdwith1() refines DTSNA1A2InotRTX()*, which returns *Valid*, meaning that there is not any event which *DTSNA1A2Ircvdwith1()* can produce but *DTSNA1A2InotRTX()* cannot. Hence, the defined attackers can prevent DTSN from reaching its goal.
   - We define *IDEALDTSNA1A2Ircvdwith1()* which is the ideal version of *DTSNA1A2Ircvdwith1()* in which the ideal versions of S, I and D are defined in the same way as in case of *IDEALDTSNA1A2()*.

- We run the model-checker and show that the assertion *IDEALA1A2SDTPIrcvdwith1() refines DTSNA1A2InotRTX()* does not hold, because *IDEALDTSNIrcvdwith1()* can produce the (input/output synch) event *DIPck.seq.ear.1* while *DTSNIrcvdwith1()* cannot.

For the SDTP protocol, let us consider the following attack scenario: S sends data packets to A1, each case A1 replaces (with probability 1) the correct ACK/NACK MACs in them to incorrect MACs and sends these modified packets to I. Node I stores some of these packets and forwards them to A2, who replaces the two MACs in each packet to the correct ones (we assume that there is a side channel between node A1 and A2). The scenario for each packet is as follows:

1. A1 receives a data packet "*seq.ear.rtx.seq1.seq2.xKack.seq3.seq4.xKnack*" from S;
2. A1 changes the correct ACK MAC "*seq1.seq2.xKack*" and NACK MACs "*seq3.seq4.xKnack*" to its MACs, generated using its key: "*seq.ear.rtx.seqA.seqA.Katt.seqA.seqA.Katt*", and sends it to node I.
3. A2 (either after receiving the packet sent by I or not) sends the correct data packet (originally sent by S) "*seq.ear.rtx.seq1.seq2.xKack.seq3.seq4.xKnack*" to D.
4. A2 forwards correctly the ACK/NACK messages that come from D to I.

This can be seen as an attack scenario because node I will never retransmit, since it stores incorrect MACs, and according to the definition of SDTP, retransmission only happens when I successfully verify the MACs. This means that the attackers can corrupt the design goal of SDTP bringing it back to the end-to-end retransmission which SDTP aimed to prevent.

The verification technique for this attack scenario using PAT is as follows:

1. In the first step, we show the (maximum) probability that node I stores at least one packet sent by A1 is larger than 0 (or some percent). We run PAT for the assertion *SDTPA1A2() reaches goal with pmax*, where *goal* is ($BufIL \geq 1$) with *BufIL* representing the number of occupied entries in the buffer.
2. In the second step
   - We define *SDTPA1A2InotRTX()*, such version of SDTP in which node I never re-transmit. The process is the same as the original *SDTPA1A2()* except that when receiving ACK/NACK messages I does not re-transmit to D. Specifically, only the output actions of the form *IDpck!seq.ear.1.seq.seq.Kack.seq.seq.Knack* for some *seq* and *ear*, are removed from *SDTPA1A2()*.
   - We define *SDTPA1A2Ircvdwith1()* in which we set the probability that I receives a packet sent by A1 to 1. Namely, we concentrate on the case when node I received at least one packet fron A1, which can happen as it is already proved in the first point.
   - We run PAT for the assertion *SDTPA1A2Ircvdwith1() refines SDTPA1A2InotRTX()*, which returns *Valid*, meaning that there is not any event which *SDTPA1A2Ircvdwith1()* can produce but *SDTPA1A2InotRTX()* cannot. Hence, the defined attackers can prevent SDTP from reaching its goal.
   - We define *IDEALSDTPA1A2Ircvdwith1()* which is the ideal version of *SDTPA1A2Ircvdwith1()* in which the ideal versions of S, I and D are defined in the same way as in case of *IDEALSDTPA1A2()*.
   - We run the model-checker and show that the assertion *IDEALA1A2SDTPIrcvdwith1() refines SDTPA1A2InotRTX()* does not hold, because *IDEALSDTPIrcvdwith1()* can produce the (input/output synch) event *DIPck.seq.ear.1.seq.seq.Kack.seq.seq.Knack* for some *seq* and *ear*, while *SDTPIrcvdwith1()* cannot.