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 Systems Security (CrySyS)
Budapest University of Technology and Economics, Hungary[1]

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

TECHNICAL REPORT

2013, January

Contents

1 Introduction

Wireless Sensor Networks (WSNs) consist low power sensor nodes aim to collect various types of data from the vicinity and send it to the sink node [1]. Numerous transport protocols specifically designed for WSN applications, requiring particularly reliable delivery and congestion control (e.g., multimedia sensor networks) have been proposed [15]. One of the latest protocols are the Distributed Transport for Sensor Networks (DTSN) [12, 14], and its secured version, the SDTP protocol [4]. 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 [3]. 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 [3]. In [3] the authors mentioned that the main vulnerabilities of reliable transport protocols for wireless sensor networks include the possibility for an attacker to replay, as well as inject fake or modified, control packets. These can lead to unrecoverable data loss; moreover, any recovery mechanism opens the doors for energy depleting attacks. In particular, using a forged or altered ACK packet, an attacker can give the sender the impression that data packets arrived safely when they may actually have been lost. This can cause the sender and the destination to become out-of-sync with respect to the status of the session. Similarly, forging or altering $NACK$ packets to trigger unnecessary retransmission can lead to denial of service or at least 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: (1) storing data packets in the buffer of sensor nodes; (2) probabilistic and real-time behavior; (3) performing cryptographic operations such as one-way hashing, digital signature, computing message authentication codes (MACs), and so on. Our main goal is to give a general method for analyse the security of WSN transport protocols, which includes either a (manual) mathematical proof technique based on formal language or an automatic model-checking technique. Moreover, we believe that our proposed methods in this paper can be used not only for verifying WSN transport protocols, but also other kind of real-time systems/protocols that perform a probabilistic behavior, and may include cryptographic primitives and operations.

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 this 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 property in a more convenient way. Furthermore, we propose an automatic verification method for this class of protocols based on a well-known model-checking framework. 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 given before, while SDTP covers all of the three points.

Specifically, the main contributions of this paper are the following:

- We propose a probabilistic timed calculus, $crypt^{prob}_{time}$, 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 $crypt_{time}^{prob}$ is inspired by the previous works [7, 8, 5] proposing solutions for each of the three discussed point, separately. In particular, $crypt_{time}^{prob}$ is a modified and combination of the well-known concepts of the applied π-calculus [7], which defines an expressive syntax and semantics supporting cryptographic primitives to analyse security protocols; the probabilistic extension of the applied π-calculus [8]; and the process calculus for timed automata proposed in [5].

We note that although in this paper the proposed $crypt_{time}^{prob}$ 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 $crypt_{time}^{prob}$ 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 [6], 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 $crypt_{time}^{prob}$ in three steps. In Section 2 we start with the introduction of the base calculus, *cryptcal*, which is the modified variant of the well-known applied π-calculus [7], designed for analysing security protocols. The extension of *cryptcal*, called $crypt_{time}$, with real-time modelling elements is given in Section 3, while in Section 4 we provide the description of $crypt_{time}^{prob}$, the probabilistic extension of $crypt_{time}$. The specifications of the DTSN and SDTP protocols in $crypt_{time}^{prob}$ can be found in Section 5 and 6, respectively. The security analysis of DTSN and SDTP, based on $crypt_{time}^{prob}$, 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 *cryptcal*: The calculus for cryptographic protocols

cryptcal is the base calculus for specifying and analysing cryptographic protocols, without supporting real-time and probabilistic systems. *cryptcal* can be seen as a modified variant of the well-known applied π-calculus [7], designed for analysing security protocols, and proving security properties of the protocols in a convenient way. Our goal is to extend *cryptcal* 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. For instance, we have to replace the process replication construct by the recursive process invocation construct, and adding the non-deterministic choice and various comparion constructs.

2.1 Syntax and informal 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

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

- c_i models 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, we assume that the binding of terms to variables is always circle free;
- e_i is a cache entry;
- Finally, f is a constructor function with arity k and is used to construct terms and to model cryptographic primitives, and messages. For instance, digital signature is modelled by the function $sign(t_1, t_2)$, where t_1 models the message to be signed and t_2 models the secret key. Complex messages are modelled by the function *tuple* of k terms: $tuple(t_1, \ldots, t_k)$, which we abbreviate as $(t_1, \ldots, t_k)$. The function symbol with arity 0 is a constant.
- int_i is the special name for modelling non-negative integers. Formally, let 0, the base element of set I, be the name that models the zero number. Let the function $inc(int_i)$ be the function that increases the integer int_i by one. Numbers 1, 2, ... are modelled by $inc(0)$, $inc(1)$, ..., 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
$P \vert Q$	parallel composition
$P + Q$	non-deterministic choice
$P[\,]Q$	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	nil
let $(x = t)$ *in* P	let 1
let $(e = t)$ *in* P	let 2

- The processes $\overline{c}\langle t\rangle.P$ 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 and binds it to x in P.
- A composition $P \mid Q$ behaves as processes P and Q running 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 non-deterministic choice $P + Q$ can behave either as P or Q in a non-deterministic way, independently from the first visible/invisible action of P and 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 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 P, and a restricted name is available only for the process within its scope.

- An another typical way of specifying infinite behavior is by using parametric recursive definitions, like in the π-calculus [13]. Here $I(y_1, \dots, y_n)$ is an identifier (or invocation) of arity n. We assume that every such identifier has a unique, possibly recursive, definition $I(x_1, \dots, x_n) \overset{def}{=} P$ where the x_i's are pairwise distinct. The intuition is that $I(y_1, \dots, y_n)$ behaves as P with each x_i replaced by y_i, respectively. It is assumed that there are finitely many such definitions. Moreover, for each $I(x_1, \dots, x_n) \overset{def}{=} P$ we require that $fn(P) \subseteq \{x_1, \dots, x_n\}$. Formally we have $I(y_1, \dots, y_n) \equiv P\{y_1/x_1, \dots, y_n/x_n\}$ if $I(x_1, \dots, x_n) \overset{def}{=} P$. We include the definition of process with parameters because we want to model the state of cache entries by specifying the value of each cache entry e_i as a process parameter, namely, $I(e_1, \dots, e_n)$. When the value of a cache has changed, we recursively invoke the process with the parameter given a new value.

- Processes $[t_i = t_j]P\ else\ Q$, $[int_i \geq int_j]P\ else\ Q$, and $[int_i > int_j]P\ else\ Q$ that 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. Processes $[t_i = t_j]P$, $[int_i \geq int_j]P$, and $[int_i > int_j]P$ are syntax sugar for shorten the process specification. They are the same as the previous three processes but Q is the nil process.

- 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 and notation of the extended process and active substitution in the applied π-calculus [7] for modelling the information the attacker (or the environment) getting to know during the system run. The definition of the *extended process* is as follows:

A, B, C ::=	extended network
P	plain network
$A \vert B$	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 networks.

- $\nu n.A$ 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 substitution $\{t/x\}$ is applied to any process that is in parallel composition with $\{t/x\}$. Intuitively,the substitution applies to *any* process that comes into contact with it. To restrict the process to which the substitution can be applied we use the variable restriction $\nu x.(\{t/x\} \,|P)$. Process P becomes $P' \mid \{t/x\}$ if P' is resulted from P after the term t has been sent to the environment. In the result process, the variable x is used to refer to term t. Active substitutions are always assumed to be cycle-free.

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:

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

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

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 of the extended process A, $A = \nu n_1 \dots n_k(\{t_1/x_1\} \mid \dots \mid \{t_n/x_n\} \mid P)$, denoted by $\varphi(A)$, is $\nu n_1 \dots n_k(\{t_1/x_1\} \mid \dots \mid \{t_n/x_n\})$. The domain of the frame $\varphi(A)$ is the set $\{x_1, \dots, x_n\}$.

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) is output in P resulting $P' \mid \{t/x\}$, t becomes available for the enviroment. Finally, let σ ranges over substitutions, we write σt for the result of applying σ to the variables of t.

2.1.1 The structural equivalence ($\equiv$)

Structural equivalence relation is defined as the least equivalence relation satisfying bound name, bound variable conversion (also called as α-conversion) and the following rules:

(Rules for Plain Processes:)

(Struct P-α)	$P \equiv_{x \leftarrow y} Q;\ P \equiv_{n_1 \leftarrow n_2} Q$
(Struct P-Par1)	$P \mid \mathbf{nil} \equiv P$
(Struct P-Par2)	$P_1 \mid P_2 \equiv P_2 \mid P_1$
(Struct P-Par3)	$(P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3)$
(Struct P-Switch)	$\nu n_1.\nu n_2.P \equiv \nu n_2.\nu n_1.P$
(Struct P-Rec)	$I(y_1, \dots, y_n) \equiv P\{y_1/x_1, \dots, y_n/x_n\}$ if $I(x_1, \dots, x_n) \overset{def}{=} P$
(Struct P-Drop)	$\nu n.\mathbf{nil} \equiv \mathbf{nil}$
(Struct P-Extr)	$\nu n.(P \mid Q) \equiv P \mid \nu n.Q$ if $n \notin fn(P)$
(Struct P-Let1)	$let\ x = t\ in\ P \equiv P\{t/x\}$
(Struct P-Let2)	$let\ e = t\ in\ P \equiv P\{t/e\}$
(Struct P-IfElse1)	$[t = t]P\ else\ Q \equiv P$
(Struct P-IfElse2)	$[t_i = t_j]P\ else\ Q \equiv Q\ (if\ t_i \neq t_j)$
(Struct P-IfElse3)	$[int_i > int_j]P\ else\ Q \equiv P\ (if\ int_i > int_j)$
(Struct P-IfElse4)	$[int_i > int_j]P\ else\ Q \equiv Q\ (if\ int_i \leq int_j)$
(Struct P-IfElse5)	$[int_i \geq int_j]P\ else\ Q \equiv P\ (if\ int_i \geq int_j)$
(Struct P-IfElse6)	$[int_i \geq int_j]P\ else\ Q \equiv Q\ (if\ int_i < int_j)$
(Struct P-If1)	$[t = t]P \equiv P$

(Struct P-If2)	$[t_i = t_j]P \equiv \mathbf{nil}$ $(if\ t_i \neq t_j)$
(Struct P-If3)	$[int_i > int_j]P\ \equiv P$ $(if\ int_i > int_j)$
(Struct P-If4)	$[int_i > int_j]P\ \equiv \mathbf{nil}$ $(if\ int_i \leq int_j)$
(Struct P-If5)	$[int_i \geq int_j]P\ \equiv P$ $(if\ int_i \geq int_j)$
(Struct P-If6)	$[int_i \geq int_j]P\ \equiv \mathbf{nil}$ $(if\ int_i < int_j)$

The meaning of each rule is the following:

- Struct P-α: P and Q are stuctural equivalent if Q can be obtained from P by renaming one or more bound names/variables in P, or vice versa. For instance, processes $(x).P$ and $(y).P$ are structural equivalent by renaming y to x. This is denoted by $\equiv_{x \leftarrow y}$.

- Struct P-Par1: The parallel composition with the nil process does not change anything, the result is the same as the original parallel composition.

- Struct P-Par2: The parallel composition is commutative.

- Struct P-Par3: The parallel composition is associative.

- Struct P-Switch: The restriction is commutative.

- Struct P-Rec: This rule is resulted directly from the definition $I(x_1, \ldots, x_n) \overset{def}{=} P$.

- Struct P-Drop: Restriction does not affect the nil process, thus, we can drop it.

- Struct P-Extrusion: We can drop the restriction from process P when P does not contain the restricted name as free name, that is, the restricted name does not occur in P.

- Struct P-Let1 and P-Let2: Both sides represent the binding of the term t to variable x (or to e) in P.

- Struct P-If1, P-If2: if the two terms are the same then the execution of P begins, while if they are distinct then the process gets stuck.

- Struct P-If3, P-If4, P-If5, P-If6: If the integer int_i is larger (or equal) than int_j then the execution of P begins, otherwise the process P gets stuck and stays idle.

Similarly as in [7], structural equivalence in closed under restriction, parallel composition, and transitivity. The structural equivalence rules for extended processes are the same as in case of the applied π- calculus [7].

2.1.2 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 communication entities. These internal operations, for instance, the verification steps made on the received data, are not visible for the outside world. 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, if the item is a general term t, then the label $\nu x.\overline{c}\langle x \rangle$ is used, after replacing the occurrence of the term t by x and wrapping the process in $\nu x.(\{t/x\}|\_)$.

(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)	$[t_i = t_j]P\ else\ Q \xrightarrow{\tau} \mathbf{nil}$ (if $t_i \neq t_j$)
(IfElse3)	$[int_i > int_j]P\ else\ Q \xrightarrow{\tau} P$ (if $int_i > int_j$)
(IfElse4)	$[int_i > int_j]P\ else\ Q \xrightarrow{\tau} \mathbf{nil}$ (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)	$[int_i \geq int_j]P\ else\ Q \xrightarrow{\tau} \mathbf{nil}$ (if $int_i < int_j$)
(If1)	$[t_i = t_j]P \xrightarrow{\tau} P$ (if $t_i = t_j$)
(If2)	$[t_i = t_j]P \xrightarrow{\tau} \mathbf{nil}$ (if $t_i \neq t_j$)
(If3)	$[int_i > int_j]P \xrightarrow{\tau} P$ (if $int_i > int_j$)
(If4)	$[int_i > int_j]P \xrightarrow{\tau} \mathbf{nil}$ (if $int_i \leq int_j$)
(If5)	$[int_i \geq int_j]P \xrightarrow{\tau} P$ (if $int_i \geq int_j$)
(If6)	$[int_i \geq int_j]P \xrightarrow{\tau} \mathbf{nil}$ (if $int_i < int_j$)
(Com)	$\overline{c}\langle t\rangle.P \mid c(x).Q \xrightarrow{\tau} P \mid Q\{t/x\}$

The silent rules Let1 and Let2 binding a variable to a term in a process; rules If1, ..., If6 check the relation of two terms or intergers. 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 rules (Com). Next, we review the rules for output/input actions borrowed from the applied π-calculus.

(Action transition rules for processes:)

(In)	$c(x).P \xrightarrow{c(t)} P\{t/x\}$
(Out)	$\overline{c}\langle u\rangle.P \xrightarrow{\overline{c}\langle u\rangle} P$
(Open)	$\dfrac{A \xrightarrow{\overline{c}\langle u\rangle} A', u \neq c}{\nu u.A \xrightarrow{\nu u.\overline{c}\langle u\rangle} A'}$
(Scope)	$\dfrac{A \xrightarrow{\alpha} A',\ u \notin \alpha}{\nu u.A \xrightarrow{\alpha} \nu u.A'}$
(Par)	$\dfrac{A \xrightarrow{\alpha} A',\ bv(\alpha) \cap fv(B) = bn(\alpha) \cap fn(B) = \emptyset}{A \mid B \xrightarrow{\alpha} A' \mid B}$
(Struct)	$\dfrac{A \equiv B, B \xrightarrow{\alpha} B', B \equiv B'}{A \xrightarrow{\alpha} A'}$

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 channel public 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 is specified by the active substitutions $\{t_1/x_1\}$ and $\{t_2/x_2\}$.

Similarly as in [7], 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 $t_1, t_2, \ldots, t_n$. We write the function as

$$tuple(t_1, t_2, \ldots, t_n)$$

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

We introduce the destructor functions **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

$$mac(t, k).$$

function that 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 $k(l_i, l_j)$.

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 and mac functions do not have any inverse counterparts because they are one-way functions.

2.2 Labeled bisimilarity

In this subsection we give the definition of labeled bisimilarity, also known in [7], 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 that is outputs so far by the process A, which are $t_1, \ldots, t_n$ in this case.

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 $A_1 \longrightarrow A_1'$, then $A_2 \longrightarrow^* A_2'$ 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 $A_2 \longrightarrow^* \xrightarrow{\alpha} \longrightarrow^* A_2'$ 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, $\longrightarrow^*$ models the sequential execution of some internal reduction steps.

3 $\boldsymbol{crypt_{time}}$: Extending *cryptcal* with timed syntax and semantics

In this subsection we propose a time extension to *cryptcal*, 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 [11], [5] in timed calculus, and based on the syntax and semantics of the well-known timed automata.

The concept of $crypt_{time}$ is based on the basic concept of timed automata, hence, the correctness of $crypt_{time}$ is 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 $v+t$ denote the valuation such that $(v+t)(x_c) = v(x_c)+t$, 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 $crypt_{time}$

$$A_t ::= A \quad | \quad \alpha^* \prec A_t \quad | \quad \phi \hookrightarrow A_t \quad | \, \phi \rhd A_t \quad | \quad \|C_R\| A_t \quad | \quad A_t^1 + A_t^2 \quad | \quad A_t^1 \; [\,] \; A_t^2 \quad | \quad (A_t^1 | A_t^2) \quad | \quad X$$

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, $\sum$, $\mathcal{C}$, $\mathcal{I}nv$, κ, $\longrightarrow$), 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;
- $\longrightarrow \subseteq \mathcal{L} \times \sum \times \Phi(\mathcal{C}) \times \mathcal{L}$ is the set of edges. We write $l \overset{act,\phi}{\longrightarrow} l'$ when (l, act, ϕ, 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 corresponds to a location l in timed automaton, such that there is an initial process A_t^0 for location l_0. The set of actions $\sum$ corresponds to the set of actions known in *cryptcal*. The set of clocks to be reset at a given location l, $\kappa(l)$, is defined by construct $\|C_R\| A_t^l$, where C_R is the set of clocks to be reset at A_t^l. The location invariant at the location l corresponds to construct $\phi \rhd A_t^l$, and the edge guard can be defined by $\phi \hookrightarrow A_t$. 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 that models internal computation of A or communication via the same channel. For instance, if A_t is $c(t).P$, where P is the plain process in *crypt*, then α^* is $c(t)$. In case of $c(x).P \mid \overline{c}\langle t\rangle.P$ α^* is τ, while in case $c(x).P + \overline{c}\langle t\rangle.P$ α^* can be either $\nu x.\overline{c}\langle x\rangle$ or $c(t)$.

- $\phi \hookrightarrow A_t$ 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 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 $\|C_R\|$ are reset within A_t. We move the clock resetting from edge to the target state like in [5].

- $A_t^1 + A_t^2$, 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 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 [13], 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) process variables (X) in them are guarded by an action (input, output, comparison) which prevents from infinite invocation. Finally, for our problem restricting X 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 [5].

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. Intuitively, a transition is defined between the source and target state, and the label of the transition composed of the actions and the time stamp (duration) of the action. When (α^*, 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 at state s until d time unit then it also stays at 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 in that in case of until, the system waits (stay idled) at least d time at 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 $\models$ on clock constraints, $\models\ \subseteq \phi(\mathcal{C})$. For each $\phi \in \phi(\mathcal{C})$ we use the shorthand $\models \phi$ iff v satisfies ϕ, $\models v(\phi)$, 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, $\forall\ v \in \mathcal{V},\ d \in \mathbb{R}^{\geq 0}$: $\models (v+d)(\phi) \Longrightarrow\ \models (v)(\phi)$. 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 [5], 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 only at the same 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, 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.*
- $fvc(A_t^1\ [\,]\ A_t^2) = fvc(A_t^1) \cup fvc(A_t^2)$
- $fvc(A_t^1 \mid A_t^2) = fvc(A_t^1) \cup fvc(A_t^2)$

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.*
- $bvc(A_t^1 \ [\,]\ A_t^2) = bvc(A_t^1) \cup bvc(A_t^2)$
- $bvc(A_t^1 \mid A_t^2) = bvc(A_t^1) \cup bvc(A_t^2)$

Recall that X is a process variable, defined as $X \overset{def}{=} P(x_1, x_2, \ldots, x_n)$, for a plain process P, and describing recursive process invocation. Since X 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 depends on the behavior of the environment.

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

9. $ncv(A_t^1 \mid A_t^2)$ 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 is because extended process A does not include clock variables; rule 2 says that the recursive process invocation of plain processes is non-conflict because plain process does not contain clock variables; rule 3 comes from the fact that action α^* is clock variable free; rule 3 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 7-9 are concerning with the cases of choices 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. $\kappa(A_t^1 \ [\,]\ A_t^2) = \kappa(A_t^1) \cup \kappa(A_t^2)$; **k8.** $\kappa(A_t^1 \mid A_t^2) = \kappa(A_t^1) \cup \kappa(A_t^2)$.

Rule k1 is because 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; the set of clocks to be reset in $\kappa(\|C_R\|\ A_t$ is C_R and the clock resets in A_t; the clock resets of choices and parallel composition is 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. $\partial\ (A_t^1\ [\,]\ A_t^2) = \partial(A_t^1) \vee \partial(A_t^2)$; **i8.** $\partial\ (A_t^1 \mid A_t^2) = \partial(A_t^1) \vee \partial(A_t^2)$.

Rule i1 says that the invariant predicate of a extended process is true because it does not include clocks; Rules i2, i3 and i4 is because there is no invariant construct defined on these processes; Rule i5 says that the invariant of process $\phi \triangleright A_t$ is predicate 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 addition, we give the rules for processes that can be connected 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. $(\phi \hookrightarrow (\alpha^* \prec (\phi' \triangleright A_t^1)\ )) + A_t^2 \overset{\alpha^*,\ \phi\wedge\phi'}{\longrightarrow} A_t^1$; **i5.** $(\phi \hookrightarrow (\alpha^* \prec (\phi' \triangleright A_t^1)\ ))\ [\,]\ A_t^2 \overset{\alpha^*,\ \phi\wedge\phi'}{\longrightarrow} A_t^1$

t6. $(\phi \hookrightarrow (\alpha^* \prec (\phi' \triangleright A_t^1)\ )) \mid A_t^2 \overset{\alpha^*,\ \phi\wedge\phi'}{\longrightarrow} A_t^1$; **t7.** $\|C_R\|(\phi \hookrightarrow (\alpha^* \prec A_t\ )) \overset{\alpha^*,\ \phi}{\longrightarrow} A_t$;

t8. $\phi' \triangleright (\phi \hookrightarrow (\alpha^* \prec A_t\ )) \overset{\alpha^*,\ \phi\wedge\phi'}{\longrightarrow} A_t$.

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.*

Proof. We have to prove that for all A_t time constructs ∂ and κ are functions, and $\partial(A_t) \in \overline{\Phi}(\mathcal{C})$. To do this we consider that $\phi^1 \wedge \phi^2 \in \overline{\Phi}(\mathcal{C})$, and $\phi^1 \vee \phi^2 \in \overline{\Phi}(\mathcal{C})$, if ϕ^1, $\phi^2 \in \overline{\Phi}(\mathcal{C})$. ☐

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

u8. $\mathcal{U}_d(A_t^1 \mid A_t^2,\ v)$ if $\mathcal{U}_d(A_t^1,\ v)$; **u9.** $\mathcal{U}_d(X,\ v)$ if $\mathcal{U}_d(P[P/X],\ v)$;

Note that in rules (u6-u8) we need to consider the commutative property of choices and parallel composition constructs to and obtaining the commutative version of these rules. For the sake of brevity we omit to give them explicitly. Next, we provide the rules concerning the timed action transitions. Rule u9 is concerning with the until predicate for (recursive) process variable X, which comes directly from the definition of recursive process invocation.

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

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

a7. $(A^1_t \mid A^2_t,\, v) \xrightarrow{\alpha^*,d}_{TTS} (A'^1_t \mid norst(A^2_t),\, v')$ if $(A^1_t,\, v) \xrightarrow{\alpha^*,d}_{TTS} (A'^1_t,\, v')$;

a8. $(X,\, v) \xrightarrow{\alpha^*,d}_{TTS} (P',\, v')$ if $(P[P/X],\, v) \xrightarrow{\alpha^*,d}_{TTS} (P',\, v')$.

The last rule is the action rules for recursive process variable X. It can be proven, based on the rules u1-u9 and a1-a8, 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 [5]. 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.** $rn(A^1_t\ [\,]\ A^2_t) = rn(A^1_t) \cup rn(A^2_t)$;

n8. $rn(A^1_t \mid A^2_t) = rn(A^1_t) \cup rn(A^2_t)$;

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-n8, 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 $A^1_t \equiv_{rn} A^2_t$ and $A'^1_t \equiv_{rn} A'^2_t$ then (i) $A^1_t + A'^1_t \equiv_{rn} A^2_t + A'^2_t$;
(ii) $A^1_t\ [\,]\ A'^1_t \equiv_{rn} A^2_t\ [\,]\ A'^2_t$;

(iii) $A_t^1 \mid A_t'^1 \equiv_{rn} A_t^2 \mid A_t'^2$;

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

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

s16. $\mathbf{true} \triangleright A_t \equiv A_t$

s17. $\phi^1 \triangleright(\phi^2 \triangleright A_t) \equiv (\phi^1 \wedge \phi^2) \triangleright A_t$

s18. $\phi \triangleright(\|C_R\|\ A_t) \equiv \|C_R\|(\phi \triangleright A_t)$, if $clock(\phi) \cap C_R = \emptyset$

s19. $\phi \triangleright (A_t^1 + A_t^2) \equiv (\phi \triangleright A_t^1) + (\phi \triangleright A_t^2)$

s20. $\phi \triangleright (A_t^1\ [\,]\ A_t^2) \equiv (\phi \triangleright A_t^1)\ [\,]\ (\phi \triangleright A_t^2)$

s21. $\|C_R\|\ A_t \equiv A_t$, if $fcv(A_t) \cap C_R = \emptyset$

s22. $\|C_R\|\ \|C_R'\|\ A_t \equiv \|C_R \cup C_R'\|\ A_t$

s23. $\|C_R\|\ (A_t^1 + A_t^2) \equiv \|C_R\|\ A_t^1 + \|C_R\|\ A_t^2$

s24. $\|C_R\|\ (A_t^1\ [\,]\ A_t^2) \equiv \|C_R\|\ A_t^1\ [\,]\ \|C_R\|\ A_t^2$.

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.

s25. $A_t^1 \mid A_t^2 \equiv A_t^2 \mid A_t^1$
s26. $\phi \hookrightarrow A_t^1 \mid A_t^2 \equiv \phi \hookrightarrow (A_t^1 \mid A_t^2)$
s27. $(A_t^1 + A_t^2) \mid A_t^3 \equiv A_t^1 \mid A_t^3 + A_t^2 \mid A_t^3$
s28. $(\|C\|\ A_t^1) \mid A_t^2 \equiv \|C\|\ (A_t^1 \mid A_t^2)$ if $C \cap fvc(A_t^2) = \emptyset$
s29. $(\partial \triangleright A_t^1) \mid A_t^2 \equiv \partial \triangleright (A_t^1 \mid A_t^2)$
s30. $(\nu k.\phi \triangleright A_t) \equiv \phi \triangleright (\nu k.A_t)$
s31. $(\nu x.\phi \triangleright A_t) \equiv \phi \triangleright (\nu x.A_t)$
s32. $(\nu k.\|C\|\ A_t) \equiv (\|C\|\nu k.A_t)$
s33. $(\nu x.\|C\|\ A_t) \equiv (\|C\|\nu x.A_t)$
s34. $(\nu k.\phi \hookrightarrow A_t) \equiv (\phi \hookrightarrow \nu k.A_t)$
s35. $(\nu x.\phi \hookrightarrow A_t) \equiv (\phi \hookrightarrow \nu x.A_t)$

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 into timed automata 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 $l_i \xrightarrow{\alpha^*,\ \phi} l_{i+1}$ in T. For the more complex target process such as $A_t^{(i+1)_1} + \ldots + A_t^{(i+1)_n}$, we have the following process definition

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

in which A_t^i corresponds to location l_i (with the appropriate resets and invariant) and the sub-process $\sum_{j=1}^{n}(\phi_j \hookrightarrow (\alpha_j^* \prec A_t^{(i+1)_j}))$ corresponds to the n edges from l_i to locations $l_{(i+1)_j}$ with labels $(\alpha_j^*,\ \phi_j)$, $1 \leq j \leq n$.

Similarly, for the target process $A_t^{(i+1)_1}\ [\ ] \ldots [\ ]\ A_t^{(i+1)_n}$ 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 sub-process $[\ ]_{j=1}^{n}(\phi_j \hookrightarrow (\alpha_j^* \prec A_t^{(i+1)_j}))$ corresponds to the edge from l_i to the location $l_{(i+1)_j}$ with label $(\alpha_j^*,\ \phi_j)$, 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 [5], and the labeled bisimilarity defined in applied π-calculus.

The Definition 7 describes the *timed labeled bisimulation* for *crypt*$_{time}$ processes.

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* $s_1 \xrightarrow{(\tau,d)}_{TTS1} s_1'$, *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* $s_1 \xrightarrow{(\alpha,d)}_{TTS1} s_1'$ *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.

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.*

From the second and third point and the axiom Delay for $U_d(s)$, we have the following consequence: Assume that the sequence of transitions from s_2 to s_2' includes k intermediate states $\{s_2^1, \ldots, s_2^k\}$, and $\sum d_i = d_0 + \ldots + d_{k+1}$, where d_0 and d_{k+1} are the time that the protocol spends at states s_2 and s_2', respectively, while each remaining d_i belongs to the state s_2^i. Then we have $\mathcal{U}_d^1(s_1)$ and $\mathcal{U}_{d_0}^2(s_2), \ldots, \mathcal{U}_{d_{k+1}}^2(s_2')$.

Recall that $\mathcal{U}_d(s)$ before the action transition $s \overset{(\alpha\cup\tau,d)}{\longrightarrow}_{TTS1} s'$ is interpreted such that performing the action takes d time unit. Not like the interpretation of $\mathcal{U}_d(s)$ in [5], we assume that there is no time for idling, only computation steps (such as verification, message composition, etc.) consuming time.

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, representing 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$_{time}^{prob}$*: The probabilistic timed calculus for cryptographic protocols

crypt$_{time}^{prob}$ is the extension of *crypt$_{time}$* with probabilistic syntax and semantics, allowing the description of timed systems equipped with a probabilistic behavior, in an intuitive and straightforward way. The definition of *crypt$_{time}^{prob}$* is inspired by the syntax and semantics of the proposed probabilistic extension of the applied π-calculus in [8], and the probabilistic automata in [5].

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 A_{pt} is an extended process in *cryptcal* with time constructs and probabilistic choice: $A_{pt}^1 \oplus_p A_{pt}^2$. Formally we have the following probabilistic timed process for *crypt$_{time}^{prob}$*:

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

Definition 8. *We extend the definition of predicates fvc and bvc with free and bound clock variables in the probabilistic choice: $fvc(A_{pt}^1 \oplus_p A_{pt}^2)$, $bvc(A_{pt}^1 \oplus_p A_{pt}^2)$. 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.*
- $fvc(A^1_{pt} \; [\,] \; A^2_{pt}) = fvc(A^1_{pt}) \cup fvc(A^2_{pt})$
- $fvc(A^1_{pt} \oplus_p A^2_{pt}) = fvc(A^1_{pt}) \cup fvc(A^2_{pt})$
- $fvc(A^1_{pt} \mid A^2_{pt}) = fvc(A^1_{pt}) \cup fvc(A^2_{pt})$

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.*
- $bvc(\phi \rhd A_{pt}) = bvc(A_{pt})$: *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 bound clock variables.*
- $bvc(A^1_{pt} \; [\,] \; A^2_{pt}) = bvc(A^1_{pt}) \cup bvc(A^2_{pt})$
- $bvc(A^1_{pt} \oplus_p A^2_{pt}) = bvc(A^1_{pt}) \cup bvc(A^2_{pt})$
- $bvc(A^1_{pt} \mid A^2_{pt}) = bvc(A^1_{pt}) \cup bvc(A^2_{pt})$

X_{pt} is a process variable, defined as $X_{pt} \overset{def}{=} P(x_1, x_2, \ldots, x_n)$, 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 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:

k9. $\kappa(A^1_{pt} \oplus_p A^2_{pt}) = \kappa(A^1_{pt}) \cup \kappa(A^2_{pt})$; **i9.** $\partial\,(A^1_{pt} \oplus_p A^2_{pt}) = \partial(A^1_{pt}) \vee \partial(A^2_{pt})$;

In addition, we give 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. $(\phi \hookrightarrow (\alpha^* \prec A^1_{pt}\ )) \mid A^2_{pt} \xrightarrow{\alpha^*,\ \phi,\ 1}_{\pi} A^1_{pt}$; if $\phi \hookrightarrow (\alpha^* \prec A^1_{pt}\ ) \xrightarrow{\alpha^*} \pi$ and $\pi(A^1_{pt}) = 1$;

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

t16. $\phi' \triangleright (\phi \hookrightarrow (\alpha^* \prec A_{pt}\ )) \xrightarrow{\alpha^*,\ \phi,\ 1}_{\pi} A_{pt}$; if $\phi' \triangleright (\phi \hookrightarrow (\alpha^* \prec A_{pt}\ )) \xrightarrow{\alpha^*} \pi$ and $\pi(A_{pt}) = 1$;

t17/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 $\pi_1(A^{1'}_{pt}) = p$;

t17/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 $\pi_2(A^{2'}_{pt}) = 1 - p$;

t18. $(\phi \hookrightarrow (\alpha^* \prec A^1_{pt}\ )) \oplus_p A^2_{pt} \xrightarrow{\alpha^*,\ \phi,\ p}_{\pi_1} A^1_{pt}$ if $\phi \hookrightarrow (\alpha^* \prec A^1_{pt}) \xrightarrow{\alpha^*}_{\pi_1} A^1_{pt}$ and $\pi_1(A^1_{pt}) = p$.

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 [8], making it also respect the operational semantics of the probabilistic timed automata [11].

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}, a clock valuation v, namely, $s = (A_{pt}, v)$. Performing either a visible α or invisible (silent) τ action consumes d time for some $d \in \mathbb{R}^{\geq 0}$. In [5] the time passage and timed action transitions are modelled with the predicate *until* $\mathcal{U}_d(s)$, and the action transition with label $s \xrightarrow{\alpha^*(d)}_{PTTS} s'$ representing that the action $\alpha^* = \alpha \cup \tau$ is performed at time d from state s, which means that the system stays at s until time d. We adopt this concept, however unlike in [5] where the idling time at s is d and executing an action takes no time, we interpret $\mathcal{U}_d(s)$ as the time for performing action α^*, and there is no idling time at s before making an action. Moreover, we extend the transition with the distribution π of the probability that is valid to the action α^*: $s \xrightarrow{\alpha^*(d),\ \pi}_{PTTS} s'$.

There are two types of transitions, an action transition with label α^*, $\xrightarrow{\alpha^*(d),\ \pi}_{PTTS}$, and the time passage transition labeled with $d' \in \mathbb{R}^{>0}$, $\xrightarrow{d'}_{PTTS}$. The scheduler F chooses nondeterministically *the distribution of action transition steps* or *(ii) to let time elapse by performing a time step*, and in case of time step also *the amount of time passed is chosen nondeterministically.* The probability of performing a transition step from a state $s = (A_{pt}, v)$ is chosen, among all the transitions enabled in s, based on the values returned by some distribution π. The probability of executing a time-step labeled with $d' \in \mathbb{R}^{>0}$ is set to 1. Formally, the enabled action transition at time d from s to s', $s \xrightarrow{\alpha^*(d),\ \pi}_{PTTS} s'$ is preceeded by a series of time step transitions via some intermediate states at which only the valuation (i.e., the 2nd parameter) is increased between two states. We denote this series of time step transitions as $\xrightarrow{\sum d_j = d}_{PTTS^*}$, where $\sum d_j = d$ means that the sum of time amounts of the time step transitions is d. Therefore, $s \xrightarrow{\alpha^*(d),\ \pi}_{PTTS} s'$ can be modelled by

$$s \xrightarrow{\sum d_j = d}_{PTTS^*} \xrightarrow{\alpha^*,\ \pi}_{PTTS} s',$$

and since the probability of time step transition is 1, the probability of $s \xrightarrow{\alpha^*(d),\ \pi}_{PTTS} s'$ is equal to the probability of transition $s \xrightarrow{\alpha^*,\ \pi}_{PTTS} s'$, according to the distribution π chosen by a scheduler F.

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 $A_{pt}^{i-1} \xrightarrow{\alpha_{i-1}^*,\ \phi_{i-1},\ p}_{\pi_{i-1}} A_{pt}^{i}$ labeled transition for probabilistic timed processes such that $v_i \models \phi_{i-1}$, $v_i \models \mathit{inv}(A_{pt}^{i-1}) \wedge \mathit{inv}(A_{pt}^{i})$, where $v_i = v_{i-1}[\kappa(A_{pt}^{i-1}) := 0] + d$, $\mathit{inv}(A_{pt}^{i-1})$ and $\kappa(A_{pt}^{i-1})$ is the invariant predicate as well as the set of clocks to be reset defined in A_{pt}^{i-1}, respectively. In addition, a time passage transition $s_{i-1} \xrightarrow{d'}_{PTTS} s_i$ is enabled if $U_{d'}(s_{i-1})$ and $v_i \models \phi_{i-1}$, $v_i \models \mathit{inv}(A_{pt}^{i-1})$, where $v_i = v_{i-1}[\kappa(A_{pt}^{i-1}) := 0] + d$. We denote the set of (time step and action) 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 $\widetilde{\alpha} = \alpha^* \cup \mathbb{R}^{\geq 0}$ by $EN(s,\ \widetilde{\alpha},\ s')$. A state $s = (A_{pt},\ v)$ is called *terminal* iff $EN(s') = \emptyset$ for all $s' = (A_{pt},\ v + d')$, $d' \in \mathbb{R}^{\geq 0}$.

A (probabilistic timed) action execution of $s_0 = (A_{pt}^0,\ v_0)$, denoted by $Exec_{s_0}$, is a finite (or infinite) sequence of 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*. 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 $\mathit{last}(e^{n_k})$ (e.g., $\mathit{last}(e^{n_k}) = s_{n_k}$).

A scheduler F defined in $\mathit{crytcal}_{time}^{prob}$ 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 $\mathit{last}(e^{n_k}) \xrightarrow{\alpha^*,\ \pi}_{PTTS} s$, for $\alpha^* \in \alpha \cup \{\tau\}$, or $\mathit{last}(e^{n_k}) \xrightarrow{d'}_{PTTS} s$, for $d' \in \mathbb{R}^{>0}$.

The execution fragments $Exec_{s_0}^F$ 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 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

$$p = \begin{cases} \dfrac{\sum_{pttr \in EN(s_{n_k-1},\ \widetilde{\alpha}_{n_k-1},\ s_{n_k})} (F(e^{n_k-1}))(pttr)}{\sum_{pttr \in EN(s_{n_k-1})} (F(e^{n_k-1}))(pttr)} & \mathit{if}\ \widetilde{\alpha}_{n_k-1} \in \alpha \cup \{\tau\} \\ 1 & \mathit{if}\ \widetilde{\alpha}_{n_k-1} \in \mathbb{R}^{>0} \end{cases}$$

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.

The operational semantics of $crypt^{prob}_{time}$ 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 following rules

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

u4. $\mathcal{U}_d(\|C_R\|\; A_{pt},\, v)$ if $\mathcal{U}_d(A_{pt},\, v[rst : C_R])$;

u5. $\mathcal{U}_d(\phi \triangleright A_{pt},\, v)$ if $\mathcal{U}_d(A_{pt},\, v) \wedge \models (v + d)(\phi)$;

u6. $\mathcal{U}_d(A^1_{pt} + A^2_{pt},\, v)$ if $\mathcal{U}_d(A^1_{pt},\, v) \vee \mathcal{U}_d(A^2_{pt},\, v)$;

u7. $\mathcal{U}_d(A^1_{pt}\; [\,]\; A^2_{pt},\, v)$ if $\mathcal{U}_d(A^1_{pt},\, v) \vee \mathcal{U}_d(A^2_{pt},\, v)$;

u8. $\mathcal{U}_d(A^1_{pt} \oplus_p A^2_{pt},\, v)$ if $\mathcal{U}_d(A^1_{pt},\, v) \vee \mathcal{U}_d(A^2_{pt},\, v)$;

u9. $\mathcal{U}_d(A^1_{pt} \mid A^2_{pt},\, v)$ if $\mathcal{U}_d(A^1_{pt},\, v) \vee \mathcal{U}_d(A^2_{pt},\, v)$;

u10. $\mathcal{U}_d(X_{pt},\, v)$ if $\mathcal{U}_d(P[P/X_{pt}],\, v)$;

Note that in rules (u6-u8) we need to consider the commutative property of choices and parallel composition constructs for obtaining the commutative version of these rules. For the sake of brevity we omit to give them explicitly. Next, we provide the rules concerning the timed action transitions. Rule u9 is concerning with the until predicate for (recursive) process variable X_{pt}, which comes directly from the definition of recursive process invocation.

a1. $(\alpha^* \prec A_{pt},\, v) \xrightarrow{\alpha^*(d),\ \pi}_{PTTS} (A_{pt},\, v + d)$ if $\alpha^* \prec A_{pt} \xrightarrow{\alpha^*,\ tt,\ 1}_{\pi} A_{pt}$;

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

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

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

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

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

a7. $(A^1_{pt}\ [\,]\ A^2_{pt},\, v) \xrightarrow{\alpha^*(d),\ \pi}_{PTTS} (A^{1'}_{pt},\, v')$ if $(A^1_{pt},\, v) \xrightarrow{\alpha^*(d),\ \pi}_{PTTS} (A^{1'}_{pt},\, v')$;

a8/a. $(A^1_{pt} \oplus_p A^2_{pt},\, v) \xrightarrow{\alpha^*(d),\ \pi_1}_{PTTS} (A^{1'}_{pt},\, v')$ if $A^1_{pt} \oplus_p A^2_{pt} \xrightarrow{\alpha^*,\ p}_{\pi_1} A^{1'}_{pt}$ and

$(A^1_{pt},\, v) \xrightarrow{\alpha^*(d),\ \pi_1}_{PTTS} A^{1'}_{pt}$;

a8/b. $(A^1_{pt} \oplus_p A^2_{pt},\, v) \xrightarrow{\alpha^*(d),\ \pi_2}_{PTTS} (A^{2'}_{pt},\, v')$ if $A^1_{pt} \oplus_p A^2_{pt} \xrightarrow{\alpha^*,\ (1-p)}_{\pi_2} A^{2'}_{pt}$ and

$(A^2_{pt},\, v) \xrightarrow{\alpha^*(d),\ \pi_2}_{PTTS} A^{2'}_{pt}$;

a9. $(A^1_{pt} \mid A^2_{pt},\, v) \xrightarrow{\alpha^*(d),\ \pi}_{PTTS} (A^{1'}_{pt} \mid norst(A^2_{pt}),\, v')$ if $(A^1_{pt},\, v) \xrightarrow{\alpha^*(d),\ \pi}_{PTTS} (A^{1'}_{pt},\, v')$;

a10. $(X_{pt},\, v) \xrightarrow{\alpha^*(d),\ \pi}_{PTTS} (P',\, v')$ if $(P[P/X_{pt}],\, v) \xrightarrow{\alpha^*(d),\ \pi}_{PTTS} (P',\, v')$.

The last rule is the action rules for recursive process variable X_{pt}. It can be proven, based on the rules u1-u10 and a1-a10, that $PTTS(A_{pt},\, v_0,\, F)$ satisfies axioms *Until* and *Delay*, hence, it is well defined.

Finally, we provide the novel bisimilarity definition, called *probabilistic timed labeled bisimilarity* for $crypt^{prob}_{time}$ which enable us to prove or refute the security of systems. 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 [7].

Given a scheduler F, $\sigma Field^F$ 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 9. ***(Probabilistic timed labeled bisimilarity for*** $crypt^{prob}_{time}$***)*** *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. Probabilistic 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* $s_2 \overset{\tau(\sum d_i),\ \pi_i}{\Longrightarrow}_{PTTS2} s'_2$ *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) $s'_1 \mathcal{R} s'_2$.

Where 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.*

From the second and third point and the axiom Delay for $U_d(s)$, we have the following consequence: Assume that the sequence of transitions from s_2 to s'_2 includes k intermediate states $\{s^1_2, \dots, s^k_2\}$, and $\sum d_i = d_0 + \dots + d_{k+1}$, where d_0 and d_{k+1} are the time that the protocol spends at states s_2 and s'_2, respectively, while each remaining d_i belongs to the state s^i_2. Then we have $\mathcal{U}^1_d(s_1)$ and $\mathcal{U}^2_{d_0}(s_2), \dots, \mathcal{U}^2_{d_{k+1}}(s'_2)$.

The arrow $\overset{\alpha}{\Longrightarrow}_{PTTS}$ is the same as $\xrightarrow{\tau}{}^{*}_{PTTS} \xrightarrow{\alpha}_{PTTS} \xrightarrow{\tau}{}^{*}_{PTTS}$, where $\xrightarrow{\tau}{}^{*}_{PTTS}$ represents a series (formally, a transitive closure) of sequential transitions $\xrightarrow{\tau}_{PTTS}$. $\sum d_i$ on $\Longrightarrow_{PTTS}$ is the sum of the time at each transition, representing the total time elapsed during the whole sequence of transitions. Note that $fn(A^2_{pt})$ 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 9 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.

5 Modelling the operation of DTSN using *crypt*$^{prob}_{time}$

5.1 DTSN - A Distributed Transport Protocol for Wireless Sensor Networks

DTSN [12, 14] 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 are that (i) 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; (ii) intermediate nodes do not store all packets but only stores with some p probability, 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. Our secure protocol that we will introduce later preserves these advantageous features of DTSN.

DTSN uses special packets to control caching and retransmissions. More specifically, there are three types of such control packets: Explicit Acknowledgement Requests (*EAR*s), Positive Acknowledgements (*ACK*s), 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

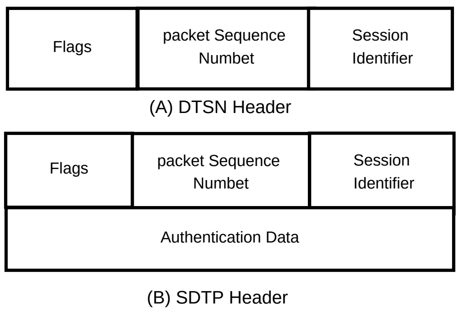

Figure 1: (A) DTSN Header. (B) SDTP Header.

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)[12, 14]. An EAR may take the form of a bit flag piggybacked on the last data packet or an independent control packet[1]. 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[12, 14]. 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 (see Fig. 1A). 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.

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.

5.2 Security issues in DTSN

Upon receiving an ACK packets, a intermediate node deletes from its cache "old" stored messages (sequence number less or equal to the sequence in the ACK packet). Thus, the intermediate node believe that those data packets have been delivered. Therefore, an attacker may lead to permanent loss of some data packets by forging or altering ACK packets. This may put the reliability service

[1]In order to decrease traffic overhead, we assume that EAR is always in the form of a bit flag piggybacked [12, 14].

provided by the protocol in danger. Moreover, an attacker can triggers unnecessary retransmission of the corresponding data 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, it is clear that such inefficiency is still undesirable. Moreover, replay ACK has harm because of the *MaxSN* while replay $NACK$ may even help the network.

Finally, 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 harmful effect. Always setting the EAR flag to 0 prevents the destination from sending an ACK or $NACK$ packet, while always setting to 1 makes the destination send control packets unnecessarily.

The security issues regarding DTSN is equivalent to any other transport layers for sensor networks which use ACK, $NACK$, EAR as control messages. For case of simplicity, we described above the security issues regarding *DTSN*. For more information about the security issues in $DTSN$ please refer to [4]. For the above reasons a secure and reliable transport protocol ($SDTP$) had to be developed.

5.3 DTSN in $crypt^{prob}_{time}$

We assume the network topology $S{-}I$ and $I{-}D$, where $-$ represents a bi-directional link, while S, I, D denote the source, an intermediate, and the destination node, respectively. Moreover, we assume that each node has three cache entries, denoted by e^s_k, e^i_k and e^d_k, $1 \le k \le 3$. The DTSN protocol on this topology can be defined by the following $crypt^{prob}_{time}$ processes: *upLayer*, *Src*, *Int*, *Dst* for specifying the behavior of the upper layer, the source, intermediate, and destination nodes. The whole DTSN protocol for the given topology is specified by the parallel composition of these four processes.

In the following, for the sake of 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}.

5.3.1 Modelling DTSN with *cryptcal*

As the first step, we define the pure (containing no time and probabilistic elements) version of each processes. In the second and third steps, we extend these processes with time and probabilistic behavior, respectively.

Process that models the (no timing) behavior of the upper layer:

$upLayer(\text{cntsq}) \overset{def}{=}$
$c_{sup}(= prvOK).hndlePck(\text{cntsq})\ [\,]\ c_{error}(= ERROR).upLayer(\text{decr(cntsq)})$
$[\,]\ c_{sessionEND}(= sEND).\mathbf{nil}\ [\,]\ c_{dup}(packet).upLayer(\text{cntsq});$

$hndlePck(\text{cntsq}) \overset{def}{=}$
$[cntsq < mxSQ]\ (\overline{c_{sup}}\langle pck, cntsq\rangle.upLayer(incr(cntsq)))\ \ else$
$[cntsq = mxSQ]\ (\overline{c_{sup}}\langle pck, cntsq\rangle.0)\ \ else\ \ upLayer(\text{cntsq});$

Process that models the (no timing) behavior of the source:

$Src(\text{s, d, apID}, e^s_{1-3}, \text{sID, earAtmp}) \overset{def}{=}$
$c_{sup}(= pck, sq).($
$fwdDt(\text{s, d, apID}, e^s_{1-3}, \text{sID, sq})$
$[\,]\ rcvACKS(\text{s, d, apID}, e^s_{1-3}, \text{sID, earAtmp})$
$[\,]\ rcvNACKS(\text{s, d, apID}, e^s_{1-3}, \text{sID, earAtmp}))\ [\,]\ c_{sessionEND}(= sEND).\mathbf{nil};$

==

$fwdDt$(s, d, apID, e^s_{1-3}, sID, sq)$\overset{def}{=}$
$[e^s_1 = E]$ *let* $e^s_1 = (s, d, apID, sID, sq)$ *in* *nxtStp1*(s, d, apID, e^s_{1-3}, sID, sq) *else*
$[e^s_2 = E]$ *let* $e^s_2 = (s, d, apID, sID, sq)$ *in* *nxtStp2*(s, d, apID, e^s_{1-3}, sID, sq) *else*
$[e^s_3 = E]$ *let* $e^s_3 = (s, d, apID, sID, sq)$ *in* *nxtStp3*(s, d, apID, e^s_{1-3}, sID, sq) *else*
$\overline{c_{error}}\langle ERROR\rangle$.*Src*(s, d, apID, e^s_{1-3}, sID, earAtmp);

---

nxtStp1(s, d, apID, e^s_{1-3}, sID, sq) $\overset{def}{=}$
$[e^s_2 = E]$ $checkAW$(s, d, apID, e^s_{1-3}, sID, sq) *else*
$[e^s_3 = E]$ $checkAW$(s, d, apID, e^s_{1-3}, sID, sq) *else*
let ear=1 in let rtx=0 in let earAtmp=1 in
$\overline{c_{si}}\langle s, d, apID, sID, sq, ear, rtx\rangle$.*Src*(s, d, apID, e^s_{1-3}, sID, earAtmp);

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

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

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

==

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

hndleACK(s, d, apID, e^s_{1-3}, sID, acknum)$\overset{def}{=}$
$[5(e^s_1) \leq acknum]$ *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 = Null) in Src(s, d, apID, e^s_{1-3}, sID, earAtmp);

---

checkE1(s, d, apID, e^s_{1-3}, sID, acknum)$\overset{def}{=}$ *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 = Null) in Src(s, d, apID, e^s_{1-3}, sID, earAtmp);

checkE2(s, d, apID, e^s_{1-3}, sID, acknum)$\overset{def}{=}$ *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 = Null) in Src(s, d, apID, e^s_{1-3}, sID, earAtmp);

checkE3(s, d, apID, e^s_{1-3}, sID, acknum)$\overset{def}{=}$ *let* $(e^s_3 = E)$ *in*
let (earAtmp = Null) in Src(s, d, apID, e^s_{1-3}, sID, earAtmp);

==

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

hndleACKNACKS1(s, d, apID, e^s_{1-3}, sID, acknum, b1) $\overset{def}{=}$
[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);

checkE1Nck1(s, d, apID, e^s_{1-3}, sID, acknum, b1) $\overset{def}{=}$ *let* ($e^s_1 = E$) *in*
[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);

checkE2Nck1(s, d, apID, e^s_{1-3}, sID, acknum, b1) $\overset{def}{=}$ *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);

checkE3Nck1(s, d, apID, e^s_{1-3}, sID, acknum, b1) $\overset{def}{=}$ *let* ($e^s_3 = E$) *in*
let (earAtmp = 0) in isSetLst(s, d, apID, e^s_{1-3}, sID, acknum, b1);

---

hndleACKNACKS2(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2) $\overset{def}{=}$
[5(e^s_1) $\leq$ *acknum*] *checkE1Nck2*(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2) *else*
[5(e^s_2) $\leq$ *acknum*] *checkE2Nck2*(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2) *else*
[5(e^s_3) $\leq$ *acknum*] *checkE3Nck2*(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2) *else*
let (earAtmp = 0) in isSet(s, d, apID, e^s_{1-3}, sID, acknum, b1).
isSetLst(s, d, apID, e^s_{1-3}, sID, acknum, b2);

checkE1Nck2(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2)$\overset{def}{=}$ *let* ($e^s_1 = E$) *in*
[5(e^s_2) $\leq$ *acknum*] *checkE2Nck2*(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2) *else*
[5(e^s_3) $\leq$ *acknum*] *checkE3Nck2*(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2) *else*
let (earAtmp = 0) in isSet(s, d, apID, e^s_{1-3}, sID, acknum, b1).
isSetLst(s, d, apID, e^s_{1-3}, sID, acknum, b2);

checkE2Nck2(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2)$\overset{def}{=}$ *let* ($e^s_2 = E$) *in*
[5(e^s_3) $\leq$ *acknum*] *checkE3Nck1*(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2) *else*
let (earAtmp = 0) in isSet(s, d, apID, e^s_{1-3}, sID, acknum, b1).
isSetLst(s, d, apID, e^s_{1-3}, sID, acknum, b2);

checkE3Nck2(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2)$\overset{def}{=}$ *let* ($e^s_3 = E$) *in*
let (earAtmp = 0) in isSet(s, d, apID, e^s_{1-3}, sID, acknum, b1).
isSetLst(s, d, apID, e^s_{1-3}, sID, acknum, b2);

---

hndleACKNACKS3(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2, b3)$\overset{def}{=}$
[5(e^s_1) $\leq$ *acknum*] *checkE1Nck3*(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2, b3) *else*
[5(e^s_2) $\leq$ *acknum*] *checkE2Nck3*(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2, b3) *else*
[5(e^s_3) $\leq$ *acknum*] *checkE3Nck3*(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2, b3) *else*
let (earAtmp = 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);

checkE1Nck3(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2, b3) $\overset{def}{=}$ *let* $(e^s_1 = E)$ *in*
  $[5(e^s_2) \leq acknum]$ *checkE2Nck3*(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2, b3) *else*
  $[5(e^s_3) \leq acknum]$ *checkE3Nck3*(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2, b3) *else*
  let $(earAtmp = 0)$ *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);

checkE2Nck3(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2, b3) $\overset{def}{=}$ *let* $(e^s_2 = E)$ *in*
  $[5(e^s_3) \leq acknum]$ *checkE3Nck3*(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2, b3) *else*
  let $(earAtmp = 0)$ *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);

checkE3Nck3(s, d, apID, e^s_{1-3}, sID, acknum, b1, b2, b3) $\overset{def}{=}$ *let* $(e^s_3 = E)$ *in*
  let $(earAtmp = 0)$ *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);

---

isSet(s, d, apID, e^s_{1-3}, sID, acknum, bt) $\overset{def}{=}$
  $[5(e^s_1) = bt]$ *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**;

rtxPck(s, d, apID, e^s_{1-3}, sID, acknum, bt) $\overset{def}{=}$
  let $(ear = 0)$ *in let* $(rtx = 1)$ *in* $\overline{c_{si}}\langle s, d, apID, sID, bt, ear, rtx\rangle$.**nil**;

isSetLst(s, d, apID, e^s_{1-3}, sID, acknum, bt) $\overset{def}{=}$
  $[5(e^s_1) = bt]$ *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*
  Src(s, d, apID, e^s_{1-3}, sID, earAtmp);

rtxPckLst(s, d, apID, e^s_{1-3}, sID, acknum, bt) $\overset{def}{=}$
  let $(ear = 1)$ *in let* $(rtx = 1)$ *in* $\overline{c_{si}}\langle s, d, apID, sID, bt, ear, rtx\rangle$.
  Src(s, d, apID, e^s_{1-3}, sID, earAtmp);

Process that models the (no timing) behavior of an intermediate node:

Int(e^i_{1-3}) $\overset{def}{=}$
  $c_{si}(s, d, apID, sID, sq, ear, rtx)$.*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**;

hndleDtI(s, d, apID, sID, sq, ear, rtx, e^i_{1-3}) $\overset{def}{=}$
  $[e^i_1 = (s, d, apID, sID, sq)]$ $\overline{c_{id}}\langle s, d, apID, sID, bt, ear, rtx\rangle$.$Int(e^i_{1-3})$ *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});

strAndFwI(s, d, apID, sID, sq, ear, rtx, e^i_{1-3}) $\overset{def}{=}$
  $[e^i_1 = E]$ *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$(e^i_{1-3}) \overset{def}{=}$
c_{idACK}(acknum).hndleACKI(s, d, apID, e^i_{1-3}, sID, acknum);

hndleACKI(s, d, apID, e^i_{1-3}, sID, acknum) $\overset{def}{=}$
$[5(e^i_1) \leq acknum]$ *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) $\overset{def}{=}$
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) $\overset{def}{=}$
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) $\overset{def}{=}$
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)$\overset{def}{=}$
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)$\overset{def}{=}$
$[5(e^i_1) \leq acknum]$ *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)$\overset{def}{=}$
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)$\overset{def}{=}$
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) $\overset{def}{=}$
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) $\overset{def}{=}$
$[5(e^i_1) = bt]$ *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* $\overline{c_{siNACK}}\langle acknum, bt\rangle.\mathbf{nil}$;

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

hndleACKNACKI2(s, d, apID, e^i_{1-3}, sID, acknum, b1, b2) $\overset{def}{=}$
$[5(e^i_1) \leq acknum]$ *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);

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

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

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

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

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

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

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

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

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

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 (no timing) behavior of the destination:

$Dst(e^d_{1-3}\text{, ackNbr, nackNbr, toRTX1, nxtsq}) \overset{def}{=}$
$\quad c_{id}(s, d, apID, sID, sq, ear, rtx).hndleDtDst\ [\ ]\ c_{sessionEND}(= sEND).\mathbf{nil};$

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

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

$sndACK1NACKDst \overset{def}{=}$
/* if nackNbr = Null 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);

$1BitInNACK \overset{def}{=}$
/* Assume dst knows it should get: 1, 2, 3. Hence num of NACK bits $\leq$ 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;

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


$goOnMissingPcktsSQE3 \overset{def}{=}$
    $[5(e^d_1) = sqnm]$ $\overline{c_{idNACK}}\langle ackNbr, sqnm\rangle.Dst(e^d_{1-3}$, ackNbr, nackNbr, toRTX1, nxtsq) else
    $[5(e^d_2) = sqnm]$ $\overline{c_{idNACK}}\langle ackNbr, sqnm\rangle.Dst(e^d_{1-3}$, ackNbr, nackNbr, toRTX1, nxtsq) else
    $[5(e^d_3) = sqnm]$ $\overline{c_{idNACK}}\langle ackNbr, sqnm\rangle.Dst(e^d_{1-3}$, ackNbr, nackNbr, toRTX1, nxtsq) else
    let sqnm=inc(sqnm) in checkMissingPcktsSQE3;

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

$goOnMissingPcktsSQG3 \overset{def}{=}$
    $[5(e^d_1) = sqnm]$ $\overline{c_{idNACK}}\langle ackNbr, sqnm\rangle.Dst(e^d_{1-3}$, ackNbr, nackNbr, toRTX1, nxtsq) else
    $[5(e^d_2) = sqnm]$ $\overline{c_{idNACK}}\langle ackNbr, sqnm\rangle.Dst(e^d_{1-3}$, ackNbr, nackNbr, toRTX1, nxtsq) else
    $[5(e^d_3) = sqnm]$ $\overline{c_{idNACK}}\langle ackNbr, sqnm\rangle.Dst(e^d_{1-3}$, ackNbr, nackNbr, toRTX1, nxtsq) else
    let sqnm=inc(sqnm) in checkMissingPcktsSQG3;

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

$check2BitMissingPckts1stBit \overset{def}{=}$
    $[sqnm > 3]$ $Dst(e^d_{1-3}$, ackNbr, nackNbr, toRTX1, nxtsq) else
    $[5(e^d_1) = sqnm]$ let toRTX1=sqnm in let sqnm=inc(sqnm) in goOn2BitMissingPckts2ndBit else
    $[5(e^d_2) = sqnm]$ let toRTX1=sqnm in let sqnm=inc(sqnm) in goOn2BitMissingPckts2ndBit else
    $[5(e^d_3) = sqnm]$ let toRTX1=sqnm in let sqnm=inc(sqnm) in goOn2BitMissingPckts2ndBit else
    let sqnm=inc(sqnm) in goOn2BitMissingPckts1stBit;
```

```
check2BitMissingPckts2ndBit $\overset{def}{=}$
    $[sqnm > 3]$ $Dst(e^d_{1-3}$, ackNbr, nackNbr, toRTX1, nxtsq$)$ else
    $[5(e^d_1) = sqnm]$ $\overline{c_{idNACK}}\langle ackNbr, toRTX1, sqnm\rangle$.
                        $Dst(e^d_{1-3}$, ackNbr, nackNbr, toRTX1, nxtsq$)$ else
    $[5(e^d_2) = sqnm]$ $\overline{c_{idNACK}}\langle ackNbr, toRTX1, sqnm\rangle$.
                        $Dst(e^d_{1-3}$, ackNbr, nackNbr, toRTX1, nxtsq$)$ else
    $[5(e^d_3) = sqnm]$ $\overline{c_{idNACK}}\langle ackNbr, toRTX1, sqnm\rangle$.
                        $Dst(e^d_{1-3}$, ackNbr, nackNbr, toRTX1, nxtsq$)$ else
    let sqnm=inc(sqnm) in goOn2BitMissingPckts2ndBit;

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

strSndUp $\overset{def}{=}$
/* Store and send to upLayer */
    $[e^d_1 = E]$ let $e^d_1 = (s, d, apID, sID, sq)$ 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 = inc(nxtsq) in
    checkEARis1 else $[e^d_2 = E]$ let $e^d_2 = (s, d, apID, sID, sq)$ in $\overline{c_{dup}}\langle s, d, apID, sID, sq\rangle$.
    let nackNbr = dec(nackNbr) in let ackNbr = nxtsq in let nxtsq = inc(nxtsq) in
    checkEARis1 else $[e^d_3 = E]$ let $e^d_3 = (s, d, apID, sID, sq)$ in $\overline{c_{dup}}\langle s, d, apID, sID, sq\rangle$.
    let nackNbr = dec(nackNbr) in let ackNbr = nxtsq in let nxtsq = inc(nxtsq) in
    checkEARis1;

strOnly $\overset{def}{=}$
/* Store and increase nackNbr, finally check EAR bit */
    $[e^d_1 = E]$ let $e^d_1 = (s, d, apID, sID, sq)$ in let nackNbr = inc(nackNbr) in checkEARis1 else
    $[e^d_2 = E]$ let $e^d_2 = (s, d, apID, sID, sq)$ in let nackNbr = inc(nackNbr) in checkEARis1 else
    $[e^d_3 = E]$ let $e^d_3 = (s, d, apID, sID, sq)$ in let nackNbr = inc(nackNbr) in checkEARis1;
```

Finally, the DTSN protocol for this topology can be defined as the parallel compisition of the three processes:

```
/* The DTSN protocol for the given topology */
DTSN(cntsq, s, d, apID, $e^i_{1-3}$, sID, earAtmp, $e^i_{1-3}$, $e^d_{1-3}$, ackNbr, nackNbr, toRTX1, nxtsq) $\overset{def}{=}$
  upLayer(cntsq) | Src(s, d, apID, $e^i_{1-3}$, sID, earAtmp) | Int($e^i_{1-3}$)
  | Dst($e^d_{1-3}$, ackNbr, nackNbr, toRTX1, nxtsq);
```

In the rest of the paper we will refer the process *DTSN*(cntsq, s, d, apID, e^i_{1-3}, sID, earAtmp, e^i_{1-3}, e^d_{1-3}, ackNbr, nackNbr, toRTX1, nxtsq) as *DTSN(params)* for brief presentation. The behavior of the whole protocol for the given topology is defined as follows:

```
/* The DTSN protocol for the given topology with variable init */
Prot(params) $\overset{def}{=}$ 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 DTSN(params);
```

5.3.2 Using *crypt*$_{time}$ to specify timed behavior of DTSN

In the second step, we will apply the time constructs of $crypt_{time}$ to define the behavior of timers in DTSN. We modify some of the processes in Section 5.3.1 as follows:

In the case of the DTSN we have to introduce two clock variables, x^{act}_c for activity timer, and x^{ear}_c for ear timer. According to the specification of the DTSN protocol [12], both clocks have the scope over all the three process, hence, to model timeout we make use of the time invariant that we define on the process *DTSN(params)*. As the result we model DTSN for the given topology as the process

$\| \; x_c^{act}, x_c^{ear} \; \| \; \{ \; x_c^{act} \leq T_{act}, x_c^{ear} \leq T_{ear} \; \} \triangleright$ *Prot(params)*;

The *Src* process then is extended with the subprocesses *actTimeOut* and *earTimeOut* that describe the behavior of the protocol in case of activity timer and ear timer expired, respectively:

The source handling Activity and EAR timer expiration:

```
$Src$(s, d, apID, $e^s_{1-3}$, sID, earAtmp) $\overset{def}{=}$
    $c_{sup}(= pck, sq).($ $\{ x_c^{act} \leq T_{act}, x_c^{ear} \leq T_{ear} \} \triangleright fwdDt$(s, d, apID, $e^s_{1-3}$, sID, sq)
                      [ ] $\{ x_c^{act} \leq T_{act}, x_c^{ear} \leq T_{ear} \} \triangleright rcvACKS$(s, d, apID, $e^s_{1-3}$, sID, earAtmp)
                      [ ] $\{ x_c^{act} \leq T_{act}, x_c^{ear} \leq T_{ear} \} \triangleright rcvNACKS$(s, d, apID, $e^s_{1-3}$, sID, earAtmp)
                     )
    [ ] $\{ x_c^{act} \leq T_{act}, x_c^{ear} \leq T_{ear} \} \triangleright c_{sessionEND}(= sEND).\mathbf{nil}$
    [ ] $\{ x_c^{act} \leq T_{act}, x_c^{ear} \leq T_{ear} \} \triangleright (x_c^{act} = T_{act}) \hookrightarrow \tau.actTimeOut$
    [ ] $\{ x_c^{act} \leq T_{act}, x_c^{ear} \leq T_{ear} \} \triangleright (x_c^{ear} = T_{ear}) \hookrightarrow \tau.earTimeOut$;
```

Activity timer expiration:
/* We note that there is no time constraint at the beginning, namely, it is true (**tt**) */

```
$actTimeOut \overset{def}{=}$
    $[lstAckNumSrc = 3] \; \overline{c_{sessionEND}} \langle sEND \rangle.\mathbf{nil}$
    [ ]
    $[lstAckNumSrc < 3] \; \overline{c_{si}} \langle EAR \rangle.$
    $\| \; x_c^{act}, x_c^{ear} \; \| \; \{ x_c^{act} \leq T_{act}, x_c^{ear} \leq T_{ear} \} \triangleright Src$(s, d, apID, $e^s_{1-3}$, sID, earAtmp);
```

EAR timer expiration:
/* We note that there is no time constraint at the beginning, namely, it is true (**tt**) */

```
$earTimeOut \overset{def}{=}$
    let earAtmp = inc(earAtmp) in
    (
      $[earAtmp > earMAX] \; \overline{c_{sessionEND}} \langle sEND \rangle.\mathbf{nil}$
      [ ]
      $[earAtmp \leq earMAX] \; \overline{c_{si}} \langle EAR \rangle.$
      $\| \; x_c^{act}, x_c^{ear} \; \| \; \{ x_c^{act} \leq T_{act}, x_c^{ear} \leq T_{ear} \} \triangleright Src$(s, d, apID, $e^s_{1-3}$, sID, earAtmp)
    );
```

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)$\overset{def}{=}$
  $c_{siACK}$(acknum).$\{ x_c^{act} \leq T_{act}, x_c^{ear} \leq T_{ear} \} \triangleright$ hndleACK(s, d, apID, $e^s_{1-3}$, sID, acknum);

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

$checkE1$(s, d, apID, $e^s_{1-3}$, sID, acknum)$\overset{def}{=}$ *let* $(e^s_1 = E)$ *in*
    $\{ x_c^{act} \leq T_{act}, x_c^{ear} \leq T_{ear} \} \triangleright [5(e^s_2) \leq acknum]$
    $\{ x_c^{act} \leq T_{act}, x_c^{ear} \leq T_{ear} \} \triangleright checkE2$(s, d, apID, $e^s_{1-3}$, sID, acknum) *else*
    $\{ x_c^{act} \leq T_{act}, x_c^{ear} \leq T_{ear} \} \triangleright [5(e^s_3) \leq acknum]$
```

$\{\ x_c^{act} \leq T_{act},\ x_c^{ear} \leq T_{ear}\ \} \triangleright$ *checkE3*(s, d, apID, e^s_{1-3}, sID, acknum) *else*
/* Here we add the resetting of the two timers on process Src */
$\{\ x_c^{act} \leq T_{act},\ x_c^{ear} \leq T_{ear}\ \} \triangleright$ *let (earAtmp = 0) in*
$\|\ x_c^{act},\ x_c^{ear}\ \|\ \{\ x_c^{act} \leq T_{act},\ x_c^{ear} \leq T_{ear}\ \} \triangleright$ *Src*(s, d, apID, e^s_{1-3}, sID, earAtmp);

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

checkE3(s, d, apID, e^s_{1-3}, sID, acknum) $\overset{def}{=}$ *let* ($e^s_3 = E$) *in*
/* Here we add the resetting of the two timers on process Src */
$\{\ x_c^{act} \leq T_{act},\ x_c^{ear} \leq T_{ear}\ \} \triangleright$ *let (earAtmp = 0) in*
$\|\ x_c^{act},\ x_c^{ear}\ \|\ \{\ x_c^{act} \leq T_{act},\ x_c^{ear} \leq T_{ear}\ \} \triangleright$ *Src*(s, d, apID, e^s_{1-3}, sID, earAtmp);

As for *rcvNACKS*, following the concept in case of *rcvACKS*, in subprocesses *isSetLst* and *rtxPckLst* we put the construct $\|\ x_c^{act},\ x_c^{ear}\ \|$ before process *Src*. According to the definition of DTSN [12], the activity timer expiration also affect 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 followed by the other processes. We note that the action τ in $(x_c^{act} = T_{act}) \hookrightarrow \tau.actTimeOut$ is the silent action that we introduce only because we want to follow the syntax of the calculus proposed in [5] which makes the process to be representable in timed automaton.

In addition to the processes above, the general rule is that the time invariant construct $\{\ x_c^{act} \leq T_{act},\ x_c^{ear} \leq T_{ear}\ \} \triangleright$ is put before each (sub)process of form $\alpha^* \prec A_t$, for some action α^*, within *DTSN(params)*. The clock resetting constructs $\|\ x_c^{act},\ x_c^{ear}\ \|$ are only added at the places discussed above, where the clocks are reset according to the specification.

5.3.3 Using $crypt^{prob}_{time}$ to specify probabilistic behavior of DTSN

In the third step, we extend the description of the DTSN protocol in $crypt_{time}$ with probabilistic choice. According to the definition of the DTSN protocol, the probabilistic choice is placed within process *Int*(e^i_{i-3}), 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*(s, d, apID, sID, sq, ear, rtx, e^i_{1-3}).

hndleDtI(s, d, apID, sID, sq, ear, rtx, e^i_{1-3}) $\overset{def}{=}$
$[e^i_1 = (s, d, apID, sID, sq)]\ \overline{c_{id}}\langle s, d, apID, sID, bt, ear, rtx\rangle.Int(e^i_{1-3})$ *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}) $\overset{def}{=}$ $\overline{c_{id}}\langle s, d, apID, sID, bt, ear, rtx\rangle.Int(e^i_{1-3})$;

we add the probabilistic choice

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

6 Modelling the operation of SDTP using $\boldsymbol{crypt}^{prob}_{time}$

6.1 SDTP - A Secure Distributed Transport Protocol for Wireless Sensor Networks

The main security solution of the SDTP protocol is as follows [4]: each (sequentially numbered) data packet is extended with an ACK MAC (Message Authentication Code) 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.

Upon receiving a data packet, the destination 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. In case an ACK packet refers to a data packet with sequence number n, the destination reveals its ACK key; similarly, when it wants to signal that this data packet is missing, the destination reveals its $NACK$ key.

Any intermediate node storing the relevant packets can verify if the ACK or $NACK$ message it receives is authentic by checking if the appropriate MAC is verified correctly with the included key. For each verification of the ACK key, the intermediate node deletes the corresponding data packets (sequence number smaller than or equal to n) from its cache. For each verification of the $NACK$ key, the intermediate node retransmits the corresponding data packet (if stored), unsets the bit, and removes the corresponding key. In case the bitmap becomes clear, the intermediate node sends an EAR message and the $NACK$ becomes an ACK message.

The ACK and $NACK$ key generation and management of SDTP is as follows: that the source and the destination share a secret which we call the session master key, and we denote it by K. From this, they both derive an ACK master key K_{ACK} and a NACK master key K_{NACK} for the 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 the pseudo-random function SessionID is the DTSN session identifier.

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

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

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.

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

To model the cryptographic primitives and operation we add the following equations into the set of equational theories:

Functions: $K(n,\ ACK)$; $K(n,\ NACK)$; $MAC(t,\ K(n,\ ACK))$;

$$\begin{aligned} \textit{Equations}\text{: } & \text{CheckMac}(MAC(t,\ K(n,\ ACK)),\ K(n,\ ACK)) = ok; \\ & \text{CheckMac}(MAC(t,\ K(n,\ NACK)),\ K(n,\ NACK)) = ok. \end{aligned}$$

where functions $K(n,\ ACK)$ and $K(n,\ NACK)$ specifies the ACK and $NACK$ per-packet keys. In order to simplify the modelling procedure, we apply the abstraction in which instead of modelling the whole key hierarchy given in [4], we assume that the per-packet keys, $K(n,\ ACK)$ and $K(n,\ NACK)$ for each valid sequence number n in one session, are not available for the attacker. The functions $MAC(t,\ K(n,\ ACK))$ and $MAC(t,\ K(n,\ NACK))$ model the MAC value computed over message t using a per-packet key. The two equations in the set of equational theories represents the verification of MAC values with the correct keys. The special constant (i.e., a function with zero arity) ok is used to model the successful verification.

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*.

```
$\textit{nxtStp1}$(s, d, apID, $e^s_{1-3}$, sID, sq) $\overset{def}{=}$
    [$e^s_2 = E$] $\textit{checkAW}$(s, d, apID, $e^s_{1-3}$, sID, sq) $\textit{else}$
    [$e^s_3 = E$] $\textit{checkAW}$(s, d, apID, $e^s_{1-3}$, sID, sq) $\textit{else}$
    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
    $\overline{c_{si}}$⟨s, d, apID, sID, sq, ear, rtx, ACKMAC, NACKMAC⟩. $\textit{Src}$(s, d, apID, $e^s_{1-3}$, sID, earAtmp);

$\textit{nxtStp2}$(s, d, apID, $e^s_{1-3}$, sID, sq) $\overset{def}{=}$
    [$e^s_3 = E$] $\textit{checkAW}$(s, d, apID, $e^s_{1-3}$, sID, sq) $\textit{else}$
    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
    $\overline{c_{si}}$⟨s, d, apID, sID, sq, ear, rtx, ACKMAC, NACKMAC⟩. $\textit{Src}$(s, d, apID, $e^s_{1-3}$, sID, earAtmp);

$\textit{nxtStp3}$(s, d, apID, $e^s_{1-3}$, sID, sq) $\overset{def}{=}$
    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
    $\overline{c_{si}}$⟨s, d, apID, sID, sq, ear, rtx, ACKMAC, NACKMAC⟩. $\textit{Src}$(s, d, apID, $e^s_{1-3}$, sID, earAtmp);

$\textit{checkAW}$(s, d, apID, $e^s_{1-3}$, sID, sq)$\overset{def}{=}$
    [$sq = AW$] ( 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
    $\overline{c_{si}}$⟨s, d, apID, sID, sq, ear, rtx, ACKMAC, NACKMAC⟩.$\textit{Src}$(s, d, apID, $e^s_{1-3}$, sID)) $\textit{else}$
    (let ear=0 in let rtx=0 in $\overline{c_{si}}$⟨s, d, apID, sID, sq, ear, rtx, ACKMAC, NACKMAC⟩.
     $\textit{Src}$(s, d, apID, $e^s_{1-3}$, 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) $\overset{def}{=}$
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 the 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) $\overset{def}{=}$
$[5(e^s_1) \leq acknum]$ $[CheckMac(6(e^s_1), ackkey) = ok]$
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) $\overset{def}{=}$
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) $\overset{def}{=}$
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.

```
hndleDtI(s, d, apID, sID, sq, ear, rtx, $e^i_{1-3}$, ackmac, nackmac) $\overset{def}{=}$
        [$e^i_1$ = (s, d, apID, sID, sq, ackmac, nackmac)]
        $\overline{c_{id}}$⟨s, d, apID, sID, bt, ear, rtx, ackmac, nackmac⟩.Int($e^i_{1-3}$) else
        [$e^i_2$ = (s, d, apID, sID, sq, ackmac, nackmac)]
        $\overline{c_{id}}$⟨s, d, apID, sID, bt, ear, rtx, ackmac, nackmac⟩.Int($e^i_{1-3}$) else
        [$e^i_3$ = (s, d, apID, sID, sq, ackmac, nackmac)]
        $\overline{c_{id}}$⟨s, d, apID, sID, bt, ear, rtx, ackmac, nackmac⟩.Int($e^i_{1-3}$) else
        strAndFwI(s, d, apID, sID, sq, ear, rtx, $e^i_{1-3}$, ackmac, nackmac);
```

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

```
strAndFwI(s, d, apID, sID, sq, ear, rtx, $e^i_{1-3}$, ackmac, nackmac) $\overset{def}{=}$
        [$e^i_1 = E$] let $e^i_1$ = (s, d, apID, sID, sq, ackmac, nackmac)
        in $\overline{c_{id}}$⟨s, d, apID, sID, bt, ear, rtx, ackmac, nackmac⟩.Int($e^i_{1-3}$) else
        [$e^i_2 = E$] let $e^i_2$ = (s, d, apID, sID, sq, ackmac, nackmac)
        in $\overline{c_{id}}$⟨s, d, apID, sID, bt, ear, rtx, ackmac, nackmac⟩.Int($e^i_{1-3}$) else
        [$e^i_3 = E$] let $e^i_3$ = (s, d, apID, sID, sq, ackmac, nackmac)
        in $\overline{c_{id}}$⟨s, d, apID, sID, bt, ear, rtx, ackmac, nackmac⟩.Int($e^i_{1-3}$) else
        $\overline{c_{id}}$⟨s, d, apID, sID, bt, ear, rtx, ackmac, nackmac⟩.Int($e^i_{1-3}$);
```

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.

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

The specifications of processes *hndleACKI*, *checkE1I*, *checkE2I* and *checkE3I* are modified similarly 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}(*acknum,b1*) is modified to c_{idNACK}(*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}(*s, d, apID, sID, sq, ear, rtx, ackmac, nackmac*). In process *hndleDtDst* the comparison [e^d_i = (*s, d, apID, sID, sq, ackmac, nackmac*)] is changed to [e^d_i = (*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*.

```
sndACK1NACKDst $\overset{def}{=}$
/* if nackNbr = Null snd ACK, if nackNbr > 0 snd NACK*/
        [nackNbr = 0] let Kack = K(ackNbr, ACK) in $\overline{c_{idACK}}$⟨ackNbr, Kack⟩.
                      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] let Kack = K(ackNbr, ACK) in let Knack1 = K(1, NACK) in
                      let Knack2 = K(2, NACK) in let Knack3 = K(3, NACK) in
                      $\overline{c_{idNACK}}$⟨ackNbr, 1, 2, 3, Kack, Knack1, Knack2, Knack3⟩.
                      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.

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

7.1 Security Analysis of the DTSN protocol

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 result in unintentional closing of a session or preventing DTSN from achieving its goal. As already mentioned before, 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 $crypt^{prob}_{time}$.

We assume that an attacker can intercept outputs of the honest nodes 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. Further, attacker(s) can send packets to its neighborhood. To describe the activity of the attacker(s) we apply the concept used in the applied π-calculus that model the presence of the attacker(s) in an implicit way, in the form of the environment. Every message that is output by the nodes taking part in the protocol is available for the environment, and although in case of sensor nodes an internal attacker (compromised node) cannot intercept the messages sent by non-neighbor nodes, it is still suitable for our attacker model in which there can be more than one attacker who can even cooperate with each other. In particular, in our model we consider a specific topology of honest nodes (source, intermediate, destination, and the number and place of the attacker(s) in the given network are not explicitly specified, but they are only revealed and to be made explicit according to a given attack scenario.

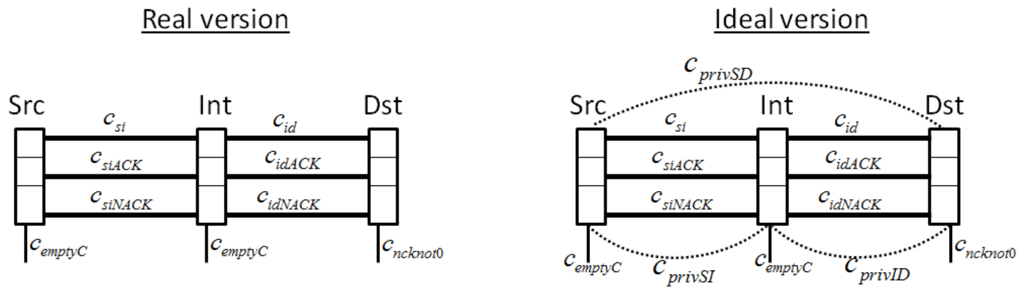

Figure 2: *The difference between the real and ideal version of the DTSN protocol.*

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 */

$Prot^{ideal}(params) \overset{def}{=}$

$\quad$ *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* $DTSN^{ideal}(params)$;

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 2 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 it should receive, by sending *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 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.

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 acknowledgement messages. Therefore, to show that DTSN is vulnerable to the modification or forging of ACK and NACK messages we prove that the operation of $Prot(params)$ is not probabilistic timed bisimilar to $Prot^{ideal}(params)$.

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:

In case of the source node (process *Src*), outputting of the constant *CacheEmptyS* is placed at the end of processes *checkE1*, *checkE2* and *checkE3*, *checkE1Nck1-3*, *checkE2Nck1-3, checkE3Nck1-3* within the subprocesses *procCacheDeletedS* and *procCacheDeletedSNck1-3*. The process *procCacheEmptyS* is invoked in which the constant *CacheEmptyS* is output whenever the number of the empty cache entries, *nbrEcache*, is 3.

With the analogous concept, in process *Int* the output of the constant *CacheEmptyI* is placed at the end of processes *checkE1I*, *checkE2I* and *checkE3I*, *checkE1NckI1-3*, *checkE2NckI1-3, checkE3NckI1-3* within the subprocesses *procCacheDeletedI* and *procCacheDeletedINck1-3*.

For the destination node we specify the process *Dst* such that the variable *nackNbr* is output via a public channel $c_{ncknot0}$ (i) whenever the number of bits (i.e., the variable *nackNbr*) within the NACK message to be sent is greater than zero; and (ii) whenever the source outputs the constant *CacheEmptyS*, with the assumption $nackNbr > 0$. This is solved by the following $crypt^{prob}_{time}$ code part in process *Dst*.

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

This process says that if $(nackNbr > 0)$ and *Dst* receives the constant *CacheEmptyS* on channel c_{deltd} 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 $\sigma_1 = \{\ldots,\ CacheEmptyS/x_i,\ nackNbr/x_j,\ \ldots\}$ 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, that is they are output right after each other, 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.

According to Definition 9 processes *Prot(params)* and *Prot^{ideal}(params)* are not probabilistic timed bisimilar because the third point of the definition is violated. Let s = (*Prot(params)*, v) and s^{ideal} = (*Prot^{ideal}(params)*, v), where v denotes the initial clock valuation (where the two clocks x_c^{act} and x_c^{ear} are reset) at both states s and s^{ideal}. Further, let s' = (*Prot'(params)*, v'), where *Prot'(params)* and v' is the process that represents undesired state, discussed above, 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 of s and s^{ideal} respectively. Morever, s_i is such a state from which there is a transition leading from s_i to s', $s_i \xrightarrow{\alpha(d),\ \pi}_{PTTS1} s'$, where α is outputting the constant *nackNbr*, (i.e., $\alpha = \nu z.\overline{c_{ncknot0}}\langle z\rangle$) where *nackNbr* is bound to z. Finally, for the proof of the first vulnerability that does not include time issues, we assume that each operation takes a constant time d.

In case of DTSN protocol 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 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_l) with $1-p$. All the other transitions are chosen with probability 1 (after the nondeterministic choice has been resolved, if any).

According to the third point of Definition 9 and the assumptions above, the following should be valid to prove the probabilistic timed bisimilarity between *Prot(params)* or *Prot^{ideal}(params)*:

1. if $s_i \xrightarrow{\alpha(d),\ \pi}_{PTTS1} s'$ and $fv(\alpha) \subseteq dom(Prot_i(params)) \wedge bn(\alpha) \cap fn(Prot_j^{ideal}(params)) = \emptyset$, then $\exists\ s^{'}_{ideal}$ such that $s^{ideal} \overset{\alpha(\sum d_j),\ \pi}{\Longrightarrow}_{PTTS2} s^{'}_{ideal}$ and
   (a) $Prob^F(s_i \xrightarrow{\alpha(d),\ \pi}_{PTTS1} s') = Prob^F(s^{ideal} \overset{\alpha(\sum d_j),\ \pi}{\Longrightarrow}_{PTTS2} s^{'}_{ideal})$;
   (b) $d = \sum d_i$
   (c) $s'\ \mathcal{R}\ s^{'}_{ideal}$.

We define function f that returns $\sum d_j$ itself. Due to the requirement $s'\ \mathcal{R}\ s^{'}_{ideal}$ the processes in the two states have to be static equivalent (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*. Note that to be static equivalence it is required that these constants are in the **same places as in the** σ_1 **or** σ_2.

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 the definition is violated. In case of *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 same action trace) in the ideal process *Prot^{ideal}(params)* when S or I (or both) empties its cache only with some zero probability. This 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 come from S, instead it generates a fake ACK message with a large base number and sends it to S which deletes its cache. Similarly, in the second topology A can make I and S free their caches. Moreover, these attack scenarios and statements are still valid in case of external attacker(s). Note that in these two attack scenarios the first point of Definition 9 is not necessarily be violated since in the ideal version both *CacheEmptyS* and *CacheEmptyI* can be output based on the normal operation of DTSN.

The following execution from state s to s' cannot be simulated by any execution trace of 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 $\{(s,\ d,\ apID,\ sID,\ 1,\ 0,\ 0)/z_2\}$. 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 $\alpha_4 = c_{siACK}(5(z_2))$ where $5(z_2)$ represents the 5th element of the packet, which is the seq number 1. The packet sent by S in the 2nd point is intercepted by attacker(s) who, instead of forwarding it, sends an ACK with base number 1 to S.

- $s_4 \xrightarrow{\alpha_5(d),\ \pi}_{PTTS1} s_5$, where $\alpha_5 = \nu z_3.\overline{c_{emptyC}}\langle z_3\rangle$ with $\{CacheEmptyS/z_3\}$. The constant *CacheEmptyS* is output upon node S erased all of its cache entries.

- $s_5 \xrightarrow{\alpha_6(d),\ \pi}_{PTTS1} s_6$, where $\alpha_6 = \nu z_4.\overline{c_{ncknot0}}\langle z_4\rangle$ with $\{nackNbr/z_4\}$. The constant *nackNbr* is output by D signalling that the number of packets to be retransmitted is greater than zero.

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 $Prob^F(s_4^{ideal} \overset{\alpha_5(d),\ \pi}{\Longrightarrow}_{PTTS2} s_5^{ideal}) > 0$, which violates the third point of Definition 9. We note that from s until s_4 $Prot^{ideal}(params)$ can simulate $Prot(params)$ via the same actions and corresponding states s^{ideal} to s_4^{ideal}.

Scenario 2 (SC-2): The violation of the first point of Definition 9 is caused by the second attack scenario where A can make both S and I empty their buffers although the destination requires retransmission of some packets. This undesired state is defined by process *Prot'(params)* and σ_2. In the topology $S - I - A - D$, first, S sends I a packet in which $(sq,\ ear,\ rtx) = (1,\ 0,\ 0)$, then I stores it and forwards to the packet to A who manipulates the packets and sends (2, 1, 0) to D. As the result, D sends a NACK message requesting the retransmission of the first packet. Meanwhile instead of forwarding the NACK message sent by the destination, A sends a fake ACK message causing I and S erase their buffers. To summarize, in this scenario the attacker can achieve that S and I empty all of their cache entries but the destination is waiting for some packets to be retransmitted.

The following execution from state s to s' cannot be simulated by any execution trace of 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 $\alpha_2 = \nu z_2.\overline{c_{si}}\langle z_2\rangle$ with $\{(s,\ d,\ apID,\ sID,\ 1,\ 0,\ 0)/z_2\}$. The source sends $(s,\ d,\ apID,\ sID,\ 1,\ 0,\ 0)$ to the intermediate node.

- $s_2 \xrightarrow{\alpha_3(d),\ \pi}_{PTTS1} s_3$, where $\alpha_3 = \nu z_3.\overline{c_{id}}\langle z_3\rangle$ with $\{(s,\ d,\ apID,\ sID,\ 1,\ 0,\ 0)/z_3\}$. Node I forwards message $(s,\ d,\ apID,\ sID,\ 1,\ 0,\ 0)$ via channel c_{id}. This output packet is available to the attacker.

- $s_3 \xrightarrow{\alpha_4(d),\ \pi}_{PTTS1} s_4$, where $\alpha_4 = c_{id}(1(z_3),\ 2(z_3),\ 3(z_3),\ 4(z_3),\ inc(5(z_3)),\ inc(6(z_3)),\ 7(z_3))$. The attacker manipulated the packet (referred as z_3) output by I, by increment the 5th and 6th elements, which are the sequence and ear bit. Afterwards, the resulted packet $(s,\ d,\ apID,\ sID,\ 2,\ 1,\ 0)$ to the destination.

- $s_4 \xrightarrow{\alpha_5(d),\ \pi}_{PTTS1} s_5$, where $\alpha_5 = \nu z_4.\overline{c_{idNACK}}\langle z_4\rangle$ with $\{(0,1)/z_4\}$. Node D sends back the NACK message requesting the retransmission of the packet with sequence number 1.

- $s_5 \xrightarrow{\alpha_6(d),\ \pi}_{PTTS1} s_6$, where $\alpha_6 = c_{idACK}(2(z_4))$. The attacker A intercepts the NACK message (z_4), and instead sends an ACK with the base number 1 to I. As the result I erases its cache.

- $s_6 \xrightarrow{\alpha_7(d),\ \pi}_{PTTS1} s_7$, where $\alpha_7 = \nu z_5.\overline{c_{emptyC}}\langle z_5\rangle$ with $\{CacheEmptyI/z_5\}$. The constant *CacheEmptyI* is output upon node I erased all cache entries.

- $s_7 \xrightarrow{\alpha_8(d),\ \pi}_{PTTS1} s_8$, where $\alpha_8 = \nu z_6.\overline{c_{ncknot0}}\langle z_6\rangle$ with $\{nackNbr/z_6\}$. The constant *nackNbr* is output by D.

- $s_8 \xrightarrow{\alpha_9(d),\ \pi}_{PTTS1} s_9$, where $\alpha_9 = \nu z_7.\overline{c_{siACK}}\langle z_7\rangle$ with $\{(1)\ /z_7\}$. Node I forwards the ACK to S.

- $s_9 \xrightarrow{\alpha_{10}(d),\ \pi}_{PTTS1} s_{10}$, where $\alpha_{10} = \nu z_8.\overline{c_{emptyC}}\langle z_8\rangle$ with $\{CacheEmptyS/z_8\}$. Node S empties all of its cache entries.

- $s_{10} \xrightarrow{\alpha_{11}(d),\ \pi}_{PTTS1} s_{12}$, where $\alpha_{12} = \nu z_9.\overline{c_{ncknot0}}\langle z_9\rangle$ with $\{nackNbr/z_9\}$. The constant *nackNbr* is output by D. Note that $s_{12} = s'$.

Based on the definition of $Prot^{ideal}(params)$ there can be the matching labeled transition trace from s^{ideal} that outputs the constants *CacheEmptyS*, *CacheEmptyI*, and *nackNbr*, however they are not placed next to each other as in case of σ_2. Therefore, according to the definition of static equivalence, from state s^{ideal} we cannot reach any state $s^{'ideal}$ such that $A^{ideal} \approx_s A^{'ideal}$, where A^{ideal} and $A^{'ideal}$ are extended processes in the states s^{ideal} and $s^{'ideal}$, respectively.

Scenario 3 (SC-3): The violation of the second point of Definition 9 is caused by the third attack scenario which is related to the timeout issue defined in DTSN. Let us consider topology $S - I - A - D$, such that during the attack scenario A sends messages that already have reached the destination before, but now with the EAR bit being always set to 1. The goal of the attacker(s) is to force D sending unnecessary ACK/NACK messages. In case of *Prot*(*params*), the attacker causes the destination to process these manipulated packets and sends back the corresponding ACK/NACK messages, which takes more time than in case of $Prot^{ideal}(params)$ for the same action trace. Note that in $Prot^{ideal}(params)$ the destination knows about what it should receive from S and I (it has been informed via the private channels c_{privSD} and c_{privID}).

Regarding process *Prot*(*params*), let the labeled action trace starting from the s to some s_k represents the following executions: (1) Upper Layer requests S to send packet *pck* with $sq = 1$; (2) S forwards *pck* to I; (3) after storing the packet, I sends it to D; (4) D received *pck* and puts it into its cache, which is represented by the state $s_k = (Prot_k(params),\ v_k)$ where $Prot_k$ is the process at s_k. The frame of $Prot_k(params)$ contains the substitution $\{pck\ /\ z_k\}$, which means that *pck* is available for the attacker. Process $Prot^{ideal}(params)$ can simulate this action trace via the same action transitions as in *Prot*(*params*), and the corresponding states from s^{ideal} to s^{ideal}_k. However, in case of $Prot_k(params)$ there is the following transition trace

- $s_k \xrightarrow{\alpha_{k+1}(d),\ \pi}_{PTTS1} s_{k+1}$, where $\alpha_{k+1} = \nu z_{k+1}.\overline{c_{id}}\langle z_{k+1}\rangle$ with $\{(1(z_k),\ 2(z_k),\ 3(z_k),\ 4(z_k),\ 5(z_k),\ inc(6(z_k)),\ 7(z_k))\ /z_{k+1}\}$. After obtaining *pck*, which is refered to as z_{k+1}, the attacker increments the 6th element of *pck*, that is, setting the ear bit from 0 to 1 and sends the modified *pck* to D.

- $s_{k+1} \xrightarrow{\alpha_{k+2}(d),\ \pi}_{PTTS1} s_{k+2}$, where $\alpha_{k+2} = c_{id}(z_{k+1})$. The destination receives the modified packets, modelled by an input action on c_{id}.

- $s_{k+2} \xrightarrow{\alpha_{k+3}(d),\ \pi}_{PTTS1} s_{k+3}$, where $\alpha_{k+3} = \nu z_{k+2}.\overline{c_{idACK}}\langle z_{k+2}\rangle$ with $\alpha_{k+2} = \{1\ /\ z_{k+2}\}$. The destination sends back the ACK packet with base number 1 to I.

To illustrate how the violation of the second point of Definition 9 is relevant in pinpointing the weaknesses of DTSN, we define $Prot^{ideal}(params)$ such that when D receives a packet with ear=1 that has not been informed by S or I (or differs from the expected message) then D sends an ACK immediately (with no any further verification steps) with some base number predefined for this purpose. Based on this specification of $Prot^{ideal}(params)$, the three transitions above can be simulated by from s_k^{ideal} with the same action transitions, and reaching the corresponding state s_{k+3}^{ideal}. However, the definition of time simulation is not valid because the total time from s_k^{ideal} to s_{k+3}^{ideal} is less than from s_k to s_{k+3} because in $Prot^{ideal}(params)$ the destination saves time by omitting the verification steps before sending the ACK.

7.2 Security Analysis of the SDTP protocol

We define the ideal version of process *ProtSDTP*(*params*), denoted by $ProtSDTP^{ideal}(params)$ and $ProtSDTP^{idealtime}(params)$, in the same concept as in $ProtDTSN^{ideal}(params)$ and porcess $ProtDTSN^{idealtime}(params)$. The only difference is that in SDTP, we define the processes *Src* and *Int* such that whenever the verification made by S and I made 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 the protocol, 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)*.

- 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 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 transition $s_2^{ideal} \xrightarrow{\alpha_3(d),\ \pi}_{PTTS2} s_3^{ideal}$. 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 and send it.

- The next transition in *ProtSDTP(params)* is $s_3 \xrightarrow{\alpha_4(d),\ \pi}_{PTTS1} s_4$ is in case of DTSN, where $\alpha_4 = c_{siACK}(t)$, describes that the attacker sends the ACK message to S with some content t. In case of DTSN it was $5(z_2)$, however, in SDTP the format of ACK required to include the correct 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 that the attacker performed on 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 $s_3^{ideal} \xrightarrow{\alpha_4(d),\ \pi}_{PTTS2} s_4^{ideal}$.

- 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 ACK message for packet 1 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 $s_4^{ideal} \xrightarrow{\alpha_5(d),\ \pi}_{PTTS2} s_5^{ideal}$ in *ProtSDTP*$_{ideal}$*(params)*.

Hence, the attack scenario (SC-1) does not work in case of SDTP protocol. However, in (SC-1) the attacker still can make the source unnecessarily handle each bogus ACK/NACK packet, consuming more time than usual. This can be formally proven by showing that *ProtSDTP*^{idealtime}*(params)* cannot simulate the transition trace $s_{4a} \xrightarrow{\tau(d),\ \pi}_{PTTS1} s_{4b}$ in *ProtSDTP(params)*, where the silent transition represents the verification of the stored ACK MAC with the ACK key included in the ACK message: defined by the code part $[CheckMac(e_i^s,\ ackkey) = \text{ok}]$, $i \in \{1, 2, 3\}$ in process *hndleACK*. Hence, based on this transition *ProtSDTP*^{idealtime}*(params)* and *ProtSDTP(params)* are not timed bisimilar.

Now we turn to examine the security of SDTP according to the Scenario-2 (SC-2).

- $s \xrightarrow{\alpha_1(d),\ \pi}_{PTTS1} s_1$ in *ProtSDTP(params)*, 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 $\alpha_2 = \nu z_2.\overline{c_{si}}\langle z_2\rangle$ with $\{(s,\ d,\ apID,\ sID,\ 1,\ 0,\ 0,\ ACK\ MAC,\ NACK\ MAC)/z_2\}$, can be simulated by the transition $s_1^{ideal} \xrightarrow{\alpha_2(d),\ \pi}_{PTTS2} s_2^{ideal}$.

- $s_2 \xrightarrow{\alpha_3(d),\ \pi}_{PTTS1} s_3$, where $\alpha_3 = \nu z_3.\overline{c_{id}}\langle z_3\rangle$ with $\{(s,\ d,\ apID,\ sID,\ 1,\ 0,\ 0,\ ACK\ MAC,\ NACK\ MAC)/z_3\}$, can be simulated by the transition $s_2^{ideal} \xrightarrow{\alpha_3(d),\ \pi}_{PTTS2} s_3^{ideal}$.

- $s_3 \xrightarrow{\alpha_4(d),\ \pi}_{PTTS1} s_4$, where $\alpha_4 = c_{id}(t)$, can be simulated by the transition $s_3^{ideal} \xrightarrow{\alpha_4(d),\ \pi}_{PTTS2} s_4^{ideal}$. Similarly as in (SC-1) $t = f_a(\mathcal{K} \cup z_3)$, differ from the correct packet sent by I. The attacker sends a packet it ables to compose based on its knowledge and computation ability.

- $s_4 \xrightarrow{\alpha_5(d),\ \pi}_{PTTS1} s_5$, where $\alpha_5 = \nu z_4.\overline{c_{badpck}}\langle z_4\rangle$ with $\{(BadPck)/z_4\}$, can be simulated by the transition $s_4^{ideal} \xrightarrow{\alpha_5(d),\ \pi}_{PTTS2} s_5^{ideal}$.

We can conclude that the attack scenario (SC-2) cannot be performed on the SDTP protocol like in DTSN. However, similarly as in (SC-1) the attacker can force the destination to perform more time consuming verification steps by sending bogus packets. This again can be proven by showing that *ProtSDTP*^{idealtime}*(params)* cannot simulate the transition trace $s_{4a} \xrightarrow{\tau(d),\ \pi}_{PTTS1} s_{4b}$ in *ProtSDTP(params)*, where the silent transition represents a MAC verification step at the destination.

In the following we prove that SDTP is not vulnerable to the attack scenario (SC-3). Again, we take the three possible transitions discussed in (SC-3):

- After obtaining the packet *pck* sent by I, which is refered to as z_{k+1}, the attacker sets the ear bit from 0 to 1 and sends this modified packet to D. Formally, this is defined by the transition $s_k \xrightarrow{\alpha_{k+1}(d),\ \pi}_{PTTS1} s_{k+1}$ in *ProtSDTP(params)*, where $\alpha_{k+1} = \nu z_{k+1}.\overline{c_{id}}\langle z_{k+1}\rangle$ with $\{(1(z_k),\ 2(z_k),\ 3(z_k),\ 4(z_k),\ 5(z_k),\ inc(6(z_k)),\ 7(z_k),\ 8(z_k),\ 9(z_k))\ /z_{k+1}\}$. Note that compared with DTSN, in SDTP z_{k+1} contains an ACK MAC and a NACK MAC that is specified by $8(z_k)$, $9(z_k)$ respectively. This transition can be simulated by $s_k^{ideal} \xrightarrow{\alpha_{k+1}(d),\ \pi}_{PTTS2} s_{k+1}^{ideal}$ in *ProtSDTP*^{ideal}*(params)*.

- $s_{k+1} \xrightarrow{\alpha_{k+2}(d),\ \pi}_{PTTS1} s_{k+2}$, where $\alpha_{k+2} = c_{id}(z_{k+1})$, can be simulated by $s^{ideal}_{k+1} \xrightarrow{\alpha_{k+2}(d),\ \pi}_{PTTS2} s^{ideal}_{k+2}$ in *ProtSDTP*^{ideal}*(params)*. The transitions say that the destination receives the modified packets, modelled by an input action on channel c_{id}.

- $s_{k+2} \xrightarrow{\alpha_{k+3}(d),\ \pi}_{PTTS1} s_{k+3}$, where $\alpha_{k+3} = \nu z_{k+2}.\overline{c_{badpck}}\langle z_{k+2}\rangle$ with $\alpha_{k+2} = \{BadPck\ /\ z_{k+2}\}$, can be simulated by $s^{ideal}_{k+2} \xrightarrow{\alpha_{k+3}(d),\ \pi}_{PTTS2} s^{ideal}_{k+3}$ in *ProtSDTP*^{ideal}(*params*). These transitions say that the destination informed its environment about the reception of an incorrect message.

Vulnerability of SDTP: The fact that in the SDTP protocol intermediate nodes do not verify the authenticity of the data packets but only store/forward them, makes SDTP be vulnerable to such attack in which the attackers send data packets with bogus (fake) MACs to intermediate nodes, and later send ACK/NACK messages corresponding to the bogus MACs. This attack is described by scenario SC-4: Let S, I, D be the source, intermediate, and destination nodes, respectively; and let $A1$ and $A2$ be the two cooperative compromised nodes. Assume that the topology is such that there are symmetrical links between $S - A1$, $A1 - I$, $I - A2$, and $A2 - D$. First, $A1$ creates a data packet *pck* containing a MAC value computed with fake ACK and NACK keys, then node I stores the packet without being able to verify the MAC values. Later, even without the presence of D, $A2$ generates fake ACK, NACK packets with the corresponding fake keys, which will match the MAC values of the stored *pck* at node I.

In $crypt^{prob}_{time}$ we can prove this vulnerability of SDTP by showing that *ProtSDTP*^{ideal}(params) and *ProtSDTP*(params) are not probabilistic timed bisimilar. The proof is based on the fact that in *ProtSDTP*^{ideal} (params) intermediate nodes will output the constant *BadControl* when they receive an unexpected packet, this cannot be simulated by *ProtSDTP*(params).

8 Automated security verification using the PAT process analysis toolkit

PAT [6] is a self-contained framework to specify and automatically verify different properties of concurrent (i.e. parallel compositions construct), real-time systems with probabilistic behavior. It provides user friendly interfaces, 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.

Currently PAT 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 [9] (the well-known process algebra based model-checker) 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 [10] (Process Meta Language), the specification language used by the SPIN [10] model-checker. PAT handles time in a tricky way, namely, instead of modeling clocks and clock resets in an explicit manner, to make model-checking, automatic verification be more effective it applies an implicit representation of time (clocks) by defining specific time expressions such as TIMEOUT, WAITUNTIL, INTERRUPT, etc.

In the following we briefly introduce four modules of PAT that we use to automatically verify the security of DTSN and SDTP. Namely, the four modules are as follows: (1) Communicating Sequential Programs (CSP#) Module, (2) Real-Time System (RTS) Module; (3) Probability CSP (PCSP) Module; (4) Probability RTS (PRTS) Module.

8.1 Communicating Sequential Programs (CSP#) Module

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

It also provides low-level constructs like variables, arrays, if-then-else, while, etc. It offers great flexibility on how to model your systems. For instance, communication among processes can be either based on shared memory (using global variables) or message passing (using asynchronous message passing or CSP-style multi-party barrier synchronization).

The high-level operators are based on the classic process algebra Communicating Sequential Processes (CSP). The main advantage of CSP# keeps the original CSP as a sub-language of CSP#, whilst 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 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, output and input action on a channel can be declared using the syntax

```
1. (declaration of channel channame): channel channame size;
2. (output of val on channame): channame!val;
3. (input a msg on channame): channame?x
```

channel is a keyword for declaring channels only, *channame* is the channel name and *size* is the channel buffer size. Channel buffer size must be greater or equal to 0. It is important that a channel with buffer size 0 sends/receives messages synchronously. This is used to model pair-wise synchronization, which involves two parties. For instance in case of *size* is 0, by puting the *channame!val* and *channame?x* in parallel, the output and input actions are performed in a synchronized form: *val* is sent and is bounded to variable x.

One of the most relevant element of the specification language in PAT is processes that is defined as an equation in the following syntax,

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

where P is the process name, *x1*,..., *xn* is an optional list of process parameters and ProcExp is a process expression. The process expression determines the internal behavior of the process. A process without parameters is written either as *P()* or P. A defined process may be referenced by its name (with the valuations of the parameters). Process referencing allows a flexible form of recursion (recursive process invocations)

PAT 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 for make debugging more straightforward and to make the returned attack traces 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. For instance, the following describes a simple vending machine which takes in a coin and dispatches a coffee every time.

```
VM() = insertcoin -> coffee -> VM();
```

An event *ev* can be a simple event orcan be attached with assignments which update global variables as in the following example, $ev\{x = x + 1;\}$ -> ***Stop***; where x is a global variable. An event may be attached with a statement block of a sequential program (which may contain local variables, if-then-else, while, math function etc.). This kind of event-prefix process is called data operation in PAT. Sequential program is considered as an atomic action, that is, no interleaving of other processes before the sequential program finishes.

PAT defines invisible events (i.e., tau event) by using keyword ***tau***, e.g., *tau* -> ***Stop***. In the tau event, statement block can still be attached. With the support of tau event, you can avoid using hiding operator to explicitly hide some visible events by name them tau events. The second way to write an invisible event is to skip the event name of a statement block, e.g., $\{x = x + 1;\}$ -> ***Stop***, which is equivalent to $tau\{x = x + 1;\}$ -> ***Stop***.

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. In PAT (as in CSP), different type of choices are defined: general choice; external choice and internal choice. General choice is resolved by any event. 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. External choice, P [$*$] Q, is resolved by the observation of a visible event (i.e., not tau event). If P performs a visible event first, then P takes control. If Q performs a visible event first, then Q takes control. Otherwise, the choice remains.

Interleaving represents two processes which run concurrently without barrier synchronization is written as P ||| Q, in which both P and Q may perform their local actions without synchronizing with each other.

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. PAT also features event *hiding* to hide events so that it is not observable by the environment. Process $P\backslash E$ where E is a set of events turns events in E to invisible ones.

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): In PAT, we support 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 instance, the following assertion asks whether the $P()$ can always eventually less than zero or not.

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

PAT supports FDR's approach for checking whether an implementation satisfies a specification or not. That is, by the notion of refinement or equivalence. Different 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()
```

8.2 Real-Time System (RTS) Module

The RTS modeule 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.

8.3 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 in the probabilistic choice (defined with the keyword ***pcase***):

```
prtsP = pcase {

             [prob1] : prtsQ1

             [prob2] : prtsQ2

                     ...

             [probn] : prtsQn

        };
```

prtsP, *prtsQ1*,..., *prtsQn* are PRTS processes which can be a normal process, a timed process, a probabilistic process or a probabilistic timed process. The choice construct says that *prtsP* can proceed as *prtsQ1*, *prtsQ2*, ..., *prtsQn* with probability *prob1*, *prob2*, ..., *probn*, respectively. The sum of the n probabilities should be 1.

For user's convenience, PAT support 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 from *prtsP* to *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 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.

8.4 On verifying DTSN using the PAT process analysis toolkit

We specify the behavior of DTSN in PAT's language with the topology $S - I - D$, where S, I and D represent the source, the intermediate and the destination node, respectively. Without losing generality and for ease the coding, reducing state space, we assume that the buffer of S and I is three and the buffer of D is four (for allowing some specific attack attempts). We choose the acknowledgement windows AW to be 2, approximately the half of the buffer size of each node. To be more precise we also include explicitly the definition of the upper layer that requests the source to send packets in sequence, and receives packets from the destination.

Between the node pairs (S, I) and (I, D) we define four symmetric channels *SI1Pck*, *SI1Ack*, *SI1Nack*, *SI1Ear*, *DI1Pck*, *DI1Ack*, *DI1Nack*, and *DI1Ear* for sending data packets, ACK, NACK, and EAR messages, respectively. In addition, we add channels between upper layer and S, D respectively. We also define the channel *EndSession* between the source and each other entities, for indicating the end of a session.

We define different constants such as the *Error* for signalling errors according to DSTN, 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 EAR packet. We assume that the probability that a packet sent by a node has lost and does not reach the addresee is, denoted by *plost*, is 10%. The probability that an intermediate node stores a packet, denoted by *pstore*, is 70%. The value of activity timer, denoted by *Tact*, is 20, while value of ear timer (*Tear*) is 10. Finally we set the maximal EAR attempt to be 5. Note that these values are only some (intuitively meaningful) example for build and running the program code, however, these values can be change easily and what important is that these values do not affect the security of DTSN, but only change the complexity, as well as the number of states during the verification.

As already mentioned, PAT 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 case of the ProVerif tool [2], which is designed for modelling and verifying security protocols, users need not to specify the behavior of attackers, because it is implicitly included during the verification algorithm. In particular in ProVerif, the attacker is specified similarly as the notion of environment in the applied π-calculus, such that it can intercept every information output by honest entities and synthesise or sending any message based on its ability. Differ from ProVerif, 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 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 between the attacker(s) and its(their) honest neighbors. We define the behavior of the attacker(s) to be as close as the attacker model assumed in ProVerif. However, in order to reduce the complexity of the verification, 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;
- the combination of these actions,

and finally it can forward the modified packets to the neighbor nodes.

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. We will follow the concept used in the manual verification in Section 7.1 and define the ideal (i.e. the specification in PAT) versions of DTSN, and examine if the real version refines the specification. Since each ideal version is designed to test specific (potential) weaknesses, in case the refinement holds it means that DTSN is not vulnerable to these weaknesses, otherwise, if the refinement does not hold the attacker can exploit these weaknesses and PAT returns an attack trace.

The ideal version of DTSN is defined in the same concept as in the manual verification in Section 7.1. In the first ideal process, defined as process *DTSNideal()* in the PAT code, there is a private channel between honest node pairs (S, I), (I, D) and (S, D). They use these channels to inform each other about the correct messages that has been sent, that is, as an addressee what should they receive. When the honest nodes receive a message that is not the expected message, they output immediately (without further actions) on a public channel *ChBadMsg* a constant *BadMsg* indicating the reception of bad message.

The information sent on the private channels is not observable by the environment, which we solve in PAT by encoding each related event as a hidden event. The second ideal process, *DTSNidealtime()* in the PAT code, is specified similarly as $ProtDTSN^{time}_{ideal}$. More precisely, *DT-SNidealtime()* is defined similarly to *DTSNideal()* but while in the second case.

In the following, we provide the definition of bad states based on the each design goal of DTSN [12]. Similarly as in the manual verification with $crypt^{time}_{prob}$:

1. The first main goal of DTSN is to provide reliable delivery of packets. DTSN supports different reliability grades in order to suit the requirements of different applications. Hence, if the attacker can achieve that the probability of delivery of some packet in a session is zero (i.e., the probability of the delivery of all packets in a session), then DTSN is not secure in presence of adversaries.

   The assertion for verifying the security of DTSN regarding the first main goal is the following:

   $$\#\textbf{\textit{define}}\ \textit{violategoal1p1}\ \ (\textit{OutBufL} == 0\ \&\&\ \textit{BufI1} == 0\ \&\&\ \textit{numNACK} > 0)$$

   where ($OutBufL == 0$) and ($BufI1 == 0$) represent the cache of S and I are emptied, but at the same time ($numNACK > 0$), which means that D has not receveid all packets. Note that this statement may appear to be a bit strict in the sense that despite ($numNACK > 0$) at the time the caches are empty, the required packets could have been retransmitted before and have not reached D yet. So there can be the lucky situation that D will eventually receive all the packets. However, this assertion still shows the weakness of DTSN since in DTSN honest nodes should empty their caches only after they receive ACK packets from D, which means that ($numNACK == 0$). We can define an another assertion that state that the caches of S and I are emptied without retransmit the required packets by D before it:

   $$\#\textbf{\textit{define}}\ \textit{violategoal1p2}\ \ (\textit{OutBufL} == 0\ \&\&\ \textit{BufI1} == 0\ \&\&\ \textit{isEARpending} == 0\ \&\&\ \textit{numNACK} > 0)$$

   which extends *violategoal1p1* with ($isEARpending == 0$) meaning that there is no EAR peding, when the caches have been deleted. Next we define the PAT codes for asking if these bad states can be reached during the DTSN:

   A1. #***assert*** *DTSN()* **reaches** *violategoal1p1*;
   A2. #***assert*** *DTSNideal()* **reaches** *violategoal1p1*;
   A3. #***assert*** *DTSN()* **reaches** *violategoal1p1 with pmax*;
   A4. #***assert*** *DTSNideal()* **reaches** *violategoal1p1 with pmax*;
   A5. #***assert*** *DTSN()* **reaches** *violategoal1p2*;
   A6. #***assert*** *DTSNideal()* **reaches** *violategoal1p2*;
   A7. #***assert*** *DTSN()* **reaches** *violategoal1p2 with pmax*;
   A8. #***assert*** *DTSNideal()* **reaches** *violategoal1p2 with pmax*;
   A9. *DTSNidealHide()* ***refines*** *DTSNHide()*.

For topology $S - I - D$ with the attacker A in the transmission range of I and D, but is not the part the route. We run the PAT model-checker with the default settings for (A1) and (A2) and get the following results: (A1) is ***Valid*** and the returned attack trace is as follows:

$S \rightarrow I$: (sq, ear, rtx) = (1, 0, 0) on channel *SI1Pck*;
$I \rightarrow D$: (sq, ear, rtx) = (1, 0, 0) on *DI1Pck* without storing,
and the packet (1, 0, 0) reaches the destination D which stores it;

$S \rightarrow I$: (sq, ear, rtx) = (2, 1, 0) on *SI1Pck*, ear = 1 since AW=2;
$I \rightarrow D$: (sq, ear, rtx) = (2, 1, 0) on *DI1Pck* without storing,
but the packet (2, 1, 0) **is lost** and does not reach the destination D;

$S \rightarrow I$: (sq, ear, rtx) = (3, 1, 0) on *SI1Pck*, ear = 1 since the buffer of S became full;
$I \rightarrow D$: (sq, ear, rtx) = (3, 1, 0) on *DI1Pck* without storing,
and the packet (3, 1, 0) reaches the destination D which stores it;

$A \rightarrow I$: then A sends the **incorrect NACK packet**, acknowledging the reception of the first 3 packets and requests the retransmission of a bogus packet 4;

$I \rightarrow S$: I received this incorrect NACK message and because its buffer is empty (because it only forwards the packets without storing) it forwards the same NACK message to S;

S: S received this incorrect NACK message and empty its buffer.

For the assertion (A2) PAT returns the text ***NOT valid*** as result, that is, *violategoal1p1* cannot be reached during *DTSNideal()* in the presence of the attacker. This means that the ideal version does not contain this vulnerability. Running PAT for (A3) and (A4) we receive the result that the maximum probability of reaching *violategoal1p1* in *DTSN()* is greater than 0, while in case of *DTSNideal()* it is 0. As we expected, the verifications of the assertions (A5 $-$ A8) give the similar results as in case of the first four, namely, (A5) is *Valid*, while (A6) is not, and the maximal probability returned in (A7) is greater than 0, and equal to 0 in (A8).

Because in PAT the refinement check between two processes is based on the equivalence of their traces of visible events, to make the verification of refinement defined in (A9) 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. For instance, in *DTSNideal()* the events that specify the communication via the private channels, as well as the equality check of the expected message and the received message are hidden from the environment, only the outputs on public channels are visible. We denote these processes as *DTSNidealHide()* and *DTSNHide()*, respectively. Now we can run the verification in PAT to examine the trace equivalent between *DTSNidealHide()* and *DTSNHide()* in (A9). PAT returns **NOT valid** because it detects a trace containing the output of constant *BadMsg* after receiving a bad NACK/ACK or data packet, which trace cannot be produced in *DTSNHide()*.

2. The second main goal of DTSN is to avoid useless wasting of energy resources through minimization of the control and retransmission overhead. DTSN attempts to achieve this by allowing intermediate nodes to store packets with a certain probability, hence, a fraction of the retransmitted packets need not to traverse the whole path from the source to the destination. Therefore, if the attacker can achieve that the packets are all deleted from the

cache of intermediate nodes, defeating the purpose of DTSN, then DTSN is insecure. We defines the bad state and assertions for verifying the second goal.

#define *violategoal2p1*
$(notExpDelCacheI = 1$ && $notExpDelCacheS = 0$ && $numNACK > 0)$.
A10. ***#assert*** *DTSN()* ***reaches*** *violategoal2p1*;
A11. ***#assert*** *DTSNideal()* ***reaches*** *violategoa12p1*;

Basically, we define the bad state as the state *violategoal2p1* where the buffer of I is emptied after receiving an ACK/NACK message from the attacker, this is modelled by (*notExpDel-CacheI* = 1), however, S did not empty its cache (*notExpDelCacheS* = 0), and the number of packets are not received by D is greater than 0. Then, we examine whether the bad state *violategoal2p1* can be reached in*DTSN()* (assertion A10) or *DTSNideal()* (A11). After running the verification in PAT, we get ***Valid*** for (A10) and ***NOT valid*** for (A11). In the first case, PAT found an attack scenario based on the topology $S - A1 - I - D$ including the second attacker $A2$ that is not the part of the route but is the neighbor of both I and D. In the returned scenario after D receives a packet with (*ear* = 1) at the moment (*numNACK* > 0),

- A_2 sends a bogus NACK message with a large base number;
- I receives this NACK message and deletes its buffer, and forwards this message to $A1$;
- $A1$ forwards the NACK message with smaller base number such that S will not erase all of its entries;
- S receives this NACK message and deletes some but not all of its cache entries.

In case of *DTSNideal()* the attacker(s) cannot perform this attack since whenever the honest nodes receive a bad NACK/ACK message they output the constant *BadMsg* instead of deleting their caches. Hence, the bad state *violategoal2p1* cannot be reached in *DTSNideal()*.

8.5 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 the *SDTPReal.prts* file that specifies the behavior of the SDTP protocol assuming the topology $S - I - D$. 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 triple ***n.ACK.K*** and ***n.NACK.K***, respectively. Where the constants *ACK* and *NACK* represent “per packet ACK key”, and “per packet NACK key”, and K represents a session master key. Then we specify the packets sent by the source node as follows: ***sq.ear.rtx.sq.sq.ACK.K.sq.sq.NACK.K***, where the first part *sq.ear.rtx* contains the packet's sequence number, the EAR and RTX bits, respectively; the second part *sq.sq.ACK.K* and the third part *sq.sq.NACK.K* represent the ACK MAC and NACK MAC respectively, computed over the packet *sq* without the EAR and RTX bits (as discussed earlier), using the ACK and NACK keys belonging to the data packet with the sequence number *sq*, *sq.ACK.K* and *sq.NACK.K* respectively.

Following the specification of SDTP, to model the fact that at the beginning the attacker(s) does not posses the per-packet ACK and NACK keys, we specify explicitly the behavior of the attacker processes such that they cannot use the master key K to construct the ACK/NACK keys of the bogus data packets. More precisely, according to the protocol attacker(s) can receive a data packet or ACK/NACK messages of the following pre-defined forms:

```
data pckt = sq.ear.rtx.sq.sq.Kack.sq.sq.Knack;

ACK msg = acknbr.acknbr.Kack

NACK msg1 = acknbr.nckb1.acknbr.Kack.nckb1.Knack;

NACK msg2 = acknbr.nckb1.nckb2.acknbr.Kack.nckb1.Knack.nckb2.Knack;

NACK msg3 = acknbr.nckb1.nckb2.nckb3.acknbr.Kack.nckb1.Knack.nckb2.Knack.
            nckb3.Knack;

NACK msg4 = acknbr.nckb1.nckb2.nckb3.nckb4.acknbr.Kack.nckb1.Knack.nckb2.Knack.
            nckb3.Knack.nckb4.Knack;
```

ACK messages include the base number *acknbr* and the corresponding per-packet ACK key *acknbr.Kack*, while NACK messages contains the ack base number, and the packets to be retransmitted, followed by the triplets that represent the corresponding per-packet ACK and NACK keys. NACK msg1 represent the case when only one packet needed to be retransmitted, and NACK msg4 is for the case of four missing packets. Attacker processes are defined such that whenever, they receive one of these messages they can use only the triplets *acknbr.Kack*, *nckb1.Knack*, *nckb2.Knack*, *nckb3.Knack*, *nckb4.Knack*, but not the key *Knack* itself. This is because we assume that the one-way hash function used in SDTP for computing per-packet keys is secure and, for example, from *acknbr.Kack* the attacker cannot deduce *Kack*. The attacker processes can include the ACK/NACK keys they received by honest nodes, or can use bogus keys that they posses or construct that are not equal to the legal keys. We assume that the attacker(s) have ACK/NACK keys of forms *acknbr.Kattack* and *nckb.Kattnack*, where *Kattack* and *Kattnack* have the similar role as *Kack*, *Knack* but is contructed by the attacker(s). Note that the attacker(s) should send the ACK/NACK keys of this 3-tuple form since this is expected by honest nodes.

The behavior of the attacker(s) can be summarized as the non-deterministic choice of each following point:

- The attacker can send (either when it receives any message or not) non-deterministically ACK and NACK messages that it constructs with its own keys *Kattack* or *Kattnack*, and some base number and bits. Further, within the NACK messages the attacker can choose non-deterministically which NACK message (among msg1,. . . , msg4) it sends to the neighbor honest nodes;

- The attacker (either when it receives any message or not) can send fake data packets with incorrect sequence number, RTX or EAR bit and MACs, to its honest neighbors.

- The attacker can be a relay node and simply forward the received packet unchaged, not affecting actively the protocol.

Note that PAT applies general purposes model-checking techniques that are not optimized to scope with the presence of the strong attacker who can do any operations on the data it has collected, which would induce a very large state space. Hence, we have to limit the behavior of the attacker such that instead of trying all the possible ack base number and the packets to be retransmitted in ACK/NACK packets, we define the following: Basically, the attacker can construct messages and forwards them based on the message it receives. More specifically,

- Before the attacker(s) receives any message from honest nodes, it can only send packets that is composed of its initial knowledge: it can non-deterministically send (1) a data packet with sequence number 5 not among the sequence number of session packets, the EAR bit is set to 1, the RTX bit is set to 0, and includes the ACK and NACK MACs computed with the attacker's ACK packet key 5.*Kattack*; (2) an ACK message with the ack base number from 5 with the corresponding ack key 5.*Kattack*.

- Whenever the attacker receives/intercepts a data packet *sq.ear.rtx.sq.sq.Kack.sq.sq.Knack* from a honest node the attacker can non-deterministically choose to (1) forward this packet unchanged; (2) to send the data packet or (non-deterministically) send an ACK with sequence and base number 5 respectively like in the first point; (3) construct and send packets from the parts of the received data packet: the clear text *sq.ear.rtx*, the ACK MAC *sq.sq.Kack*, and the NACK MAC *sq.sq.Knack*. More specifically, the attacker can compose and send the data packet *sq.ear.rtx.sq.sq.Kattack.sq.sq.Kattnack* using the clear text part and its computed MACs. The attacker can also send packet *sq.ear'.rtx.sq.sq.Kattack.sq.sq.Kattnack*, *sq.ear.rtx'.sq.sq.Kattack.sq.sq.Kattnack*, or *sq.ear'.rtx'.sq.sq.Kattack.sq.sq.Kattnack*, where ear' and rtx' are the negation of ear and rtx, respectively.

- When the attacker receives/intercepts an ACK message *acknbr.acknbr.Kack* from a honest node the attacker posses its parts *acknbr* and *acknbr.Kack*, and it can non-deterministically choose to (1) forward the message unchanged; (2) to send the data packet or (non-deterministically) send an ACK with sequence and base number 5 respectively like in the first point; (3) to send the data packets *acknbr.ear.rtx.acknbr.acknbr.Kattack.acknbr.acknbr.Kattnack*, with *ear*, *rtx* $\in \{0, 1\}$; (4) to send the NACK message *0.acknbr.acknbr.Knack*, requesting the retransmission of packet acknbr;

- When the attacker receives/intercepts an NACK message with one nack bit *acknbr.nckb1.acknbr.Kack.nckb1.Knack* from a honest node, it posseses the parts *acknbr*, *nckb1*, *acknbr.Kack*, *nckb1.Knack*. Based on this knowledge it can send the following messages: (1) the received NACK message unchanged; (2) replay the data packets *acknbr.ear.rtx.acknbr.acknbr.Kack.acknbr.acknbr.Knack* if it has obtained this data packet before; (3) it can send packets with different values of ear/rtx bits, *acknbr.ear.rtx.acknbr.acknbr.Kack.acknbr.acknbr.Knack* if the attacker posseses the NACK key corresponding to seqnum *acknbr*, which can happen if it has obtained the NACK packet referred to *acknbr* before; (4) it sends ACK messages *acknbr.acknbr.Kack*, *nckb1.nckb1.Kack*, 5.5.*Kack*; (5) it sends NACK message based on the received NACK: *0.nckb1.nckb1.Knack* and *0.acknbr.acknbr.Knack* or 0.*acknbr.nckb1.acknbr.Knack.nckb1.nckb1.Knack* if the attacker has obtained before the NACK key corresponding to seqnum *acknbr*.

- When the attacker receives/intercepts an NACK message with two, three and four nack bits *acknbr.nckb1.nckb2.acknbr.Kack.nckb1.Knack.nckb2.Knack*,

  acknbr.nckb1.nckb2.nckb3.acknbr.Kack.nckb1.Knack.nckb2.Knack.nckb3.Knack and

  acknbr.nckb1.nckb2.nckb3.nckb4.acknbr.Kack.nckb1.Knack.nckb2.Knack.nckb3.Knack.nckb4.Knack, respectively, from a honest node, it creates and sends data packets, ACK and NACK messages in the similar concept as in the third point.

Now we turn to discuss the automatic verification of SDTP in PAT. First, to see if SDTP reaches its design goals in hostile environment we define the ideal version of SDTP, defined with process *SDTPideal()*, in the same concept as in the case of DTSN. Namely, there is a private channel between honest node pairs (S, I), (I, D) and (S, D) to inform about the correct messages. In case of improper message are received, the constant *BadMsg* is output immediately on the public channel *ChBadMsg*. Otherwise, *SDTPideal()* behaves similarly as *SDTP()*.

Basically, the design goal of SDTP is to make DTSN achieve its goals in a hostile environment. Hence, we examine every assertions defined in case of DTSN.

1. The defined PAT codes for asking if these bad states can be reached for SDTP case is as follows:

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. #***assert*** *SDTP()* ***reaches*** *violategoal1p2*;
B6. #***assert*** *SDTPideal()* ***reaches*** *violategoal1p2*;
B7. #***assert*** *SDTP()* ***reaches*** *violategoal1p2 with pmax*;
B8. #***assert*** *SDTPideal()* ***reaches*** *violategoal1p2 with pmax*;
B9. *SDTPidealHide()* ***refines*** *SDTPHide()*.

Recall that (B1) reaching the bad state defined by *violategoal1p1*, for verifying the security of DTSN regarding the first main goal, is Valid in case of DTSN. We run PAT to model-check (B1) 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. Similarly, reaching the assertion *violategoal1p2* (B5), which is Valid in DTSN, is ***Not valid*** in SDTP. Assertions (B2) and (B6) also result in ***Not valid***. Assertions (B3), (B4), (B7) and (B8) return 0 as the maximal probability *pmax*. Finally, checking (B9) 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.

9 Conclusion

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 property in a more straightforward 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.

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 the concurrent, non-deterministic, real time, and probabilistic behavior of systems in a convenient way. However, in the 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.

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] I. F. Akyildiz, S. Weilian, Y. Sankarasubramaniam, and E. Cayirci, *Survey on sensor networks*, Computer Networks **38** (2002), no. 4, 393–422.

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

[3] 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.

[4] 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.

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

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

[7] 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.

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

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

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

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

[12] 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.

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

[14] 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.

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