Hindawi Publishing Corporation
ISRN Sensor Networks
Volume 2014, Article ID 891467, 20 pages
http://dx.doi.org/10.1155/2014/891467

Research Article

On Formal and Automatic Security Verification of WSN Transport Protocols

Vinh Thong Ta,[1] Levente Buttyán,[1,2] and Amit Dvir[3]

[1] *Laboratory of Cryptography and Systems Security (CrySyS), Budapest University of Technology and Economics, Budapest 1117, Hungary*
[2] *MTA-BME Information Systems Research Group, Magyar tudósok körútja 2, Budapest 1117, Hungary*
[3] *Computer Science Department, College of Management Academic Studies, 7 Yitzhak Rabin Boulevard, 75190 Rishon LeZion, Israel*

Correspondence should be addressed to Levente Buttyán; buttyan@crysys.hu

Received 21 October 2013; Accepted 17 December 2013; Published 4 March 2014

Academic Editors: J. Li, S. Srinivasan, and Y. Yu

We address the problem of formal and automated security verification of transport protocols for wireless sensor networks (WSN) 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, allowing for studying real-time, probabilistic, and cryptographic issues at the same time. Hence, it can be used to verify systems that involve these three properties in a convenient way. In addition, we propose an automatic verification method, based on the well-known PAT process analysis toolkit, for this class of protocols. For demonstration purposes, we apply the proposed manual and automatic proof methods for verifying the security of DTSN and SDTP, which are two of the recently proposed WSN transport protocols.

1. Introduction

Numerous transport protocols have been proposed specifically designed for applications of wireless sensor networks (WSN), requiring particularly reliable delivery and congestion control (e.g., multimedia sensor networks) [1]. Two of the latest protocols are the distributed transport for sensor networks (DTSN) [2] and its secured version, the secure distributed transport protocol for sensor networks (SDTP) [3]. In DTSN and SDTP the intermediate nodes can cache the packets with some probability and retransmit them upon request, providing reliable transmission, energy efficiency, and distributed functionality.

Unfortunately, existing transport protocols for WSNs (including DTSN) do not include sufficient security mechanisms or totally ignore the security issue. Hence, many attacks have been found against existing WSN transport protocols [4]. Broadly speaking, these attacks can be classified into two groups: attacks against reliability and energy depleting attacks. Reliability attacks aim to mislead the nodes so that loss of a data packet remains undetected. In the case of energy depleting attacks, the goal of the attacker is to perform energy-intensive operations in order to deplete the nodes' batteries [4]. In particular, using a fake or altered acknowledgment message, an attacker can give the sender the impression that data packets arrived safely when they may actually have been lost. Similarly, forging or altering negative acknowledgment packets to trigger unnecessary retransmission can lead to faster draining of the node's batteries. While futile retransmissions do not directly harm the reliability of service, they are still undesirable. Systematic mathematical and automated methods are needed for finding the weaknesses in these protocols; however, this is a very hard task due to their complexity. This served as a motivation and challenge for our work, to which we proposed solutions.

In this paper, we address the problem of formal and automated security verification of WSN transport protocols, which typically consist of the following behavioral characteristics: (b1) storing data packets in the buffer of intermediate sensor nodes; (b2) probabilistic and real-time behavior; (b3) performing cryptographic operations such as one-way hashing, computing message authentication codes (MACs); and so on. We propose a formal and an automated verification method, based on the application of process algebra and a model checking framework, respectively. For demonstration purposes, we apply the proposed methods for specifying and verifying the security of the DTSN and the SDTP protocols, which are representative in the sense that DTSN involve the first two behavioral characteristics (b1-b2), while SDTP covers all of the three points (b1–b3). Specifically, the main contributions of this paper are the followings.

(i) We propose a probabilistic timed calculus, called $crypt^{prob}_{time}$, for cryptographic protocols. To the best of our knowledge, this is the first of its kind in the sense that it combines the following three features: (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 systems that perform probabilistic behavior. The basic concept of $crypt^{prob}_{time}$ is inspired by previous pieces of work [5–7] proposing solutions separately for each of the three discussed points. In particular, $crypt^{prob}_{time}$ is derived from the applied π-calculus [5], which defines an expressive syntax and semantics supporting cryptographic primitives to analyse security protocols; a probabilistic extension of the applied π-calculus [6]; and a process calculus for timed automata proposed in [7].

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

(ii) Using $crypt^{prob}_{time}$ we specify the behavior of the DTSN and SDTP protocols. We propose the novel definition of *weak probabilistic timed bisimilarity* and used it to prove the weaknesses of DTSN and SDTP, as well as the security of SDTP against some attacks.

(iii) We provide an approach for the automatic security verification of the DTSN and SDTP protocols with the PAT process analysis toolkit [8], which is a powerful general purpose model checking framework. To the best of our knowledge PAT has not been used for this purpose before; however, in this paper we show that the power of PAT can be used to check some interesting security properties defined for these systems/protocols.

The structure of the paper is as follows: note that, instead of putting into a separate section, we discuss the related pieces of work inside each section, where we compare them with our methods. In Section 2, we start with the introduction of DTSN and SDTP. In Section 3, we discuss the base calculus, *crypt*, which is a modified variant of the well-known applied π-calculus [5], designed for analysing security protocols. The extension of *crypt*, called $crypt^{prob}_{time}$, with timing and probabilistic modelling elements is given in Section 4. The security analysis of DTSN and SDTP, based on $crypt^{prob}_{time}$, is provided in Section 5. The well-known model checking framework PAT and automatic verification of SDTP are given in Section 6. We also verified the DTSN protocol in PAT and found attacks against it; however, due to lack of space we do not discuss it in the paper. Interested readers can find more details in our longer report [11]. Finally, we conclude the paper and give an outlook on future pieces of work in Section 7.

2. The DTSN and SDTP Protocols

2.1. DTSN: Distributed Transport for Sensor Networks. DTSN [2] is a reliable transport protocol developed for sensor networks where intermediate nodes between the source and the destination of a data flow cache data packets in a probabilistic manner such that they can retransmit them upon request. The main advantages of DTSN compared to a transport protocol that uses a fully end-to-end retransmission mechanism is that it allows intermediate nodes to cache and retransmit data packets; hence, the average number of hops that a retransmitted data packet must travel is smaller than the length of the route between the source and the destination. Intermediate nodes do not store all packets but only store packets with some probability p, which makes it more efficient. Note that, in the case of a fully end-to-end reliability mechanism, where only the source is allowed to retransmit lost data packets, retransmitted data packets always travel through the entire route from the source to the destination. Thus, DTSN improves the energy efficiency of the network compared to a transport protocol that uses a fully end-to-end retransmission mechanism.

DTSN uses special packets to control caching and retransmissions. More specifically, there are three types of such control packets: Explicit Acknowledgment Requests (*EARs*), Positive Acknowledgments (*ACKs*), and Negative Acknowledgments (*NACKs*). The source sends an *EAR* packet after the transmission of a certain number of data packets or when its output buffer becomes full or when the application has not requested the transmission of any data during a predefined timeout period or due to the expiration of the *EAR* timer (*EAR_timer*).

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

the session will be terminated or reset. The EAR timer is launched whenever an EAR packet or a data packet with the EAR bit set is sent.

An *EAR* may take the form of a bit flag piggybacked on the last data packet or an independent control packet. An *EAR* is also sent by an intermediate node or the source after retransmission of a series of data packets, piggybacked on the last retransmitted data packet [2]. Upon receipt of an *EAR* packet the destination sends an *ACK* or a *NACK* packet, depending on the existence of gaps in the received data packet stream. An *ACK* refers to a data packet sequence number *n*, and it should be interpreted such that all data packets with sequence number smaller than or equal to *n* were received by the destination. A *NACK* refers to a base sequence number *n* and it also contains a bitmap, in which each bit represents a different sequence number starting from the base sequence number *n*. A *NACK* should be interpreted such that all data packets with sequence number smaller than or equal to *n* were received by the destination and the data packets corresponding to the set bits in the bitmap are missing.

Within a session, data packets are sequentially numbered. The Acknowledgment 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, 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.

Security Issues in DTSN. Upon receiving an *ACK* packet, intermediate nodes delete from their cache the stored messages whose sequence number is less than or equal to the sequence number in the *ACK* packet, because the intermediate nodes believe that acknowledged packets have been delivered successfully. Therefore, an attacker may cause permanent loss of some data packets by forging or altering *ACK* packets. This may put the reliability service provided by the protocol in danger. Moreover, an attacker can trigger unnecessary retransmission of the corresponding data packets by either setting bits in the bit map of the *NACK* packets or forging/altering *NACK* packets. Any unnecessary retransmission can lead to energy consumption and interference. Note that unnecessary retransmissions do not directly harm the reliability, but it is clear that such inefficiency is still undesirable.

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

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

In particular, the security solution of SDTP works as follows [3]: each data packet is extended with an *ACK* MAC and a *NACK* MAC, which are computed over the whole packet with two different keys, an *A*CK key (K_{ACK}) and a *NACK* key (K_{NACK}). Both keys are known only to the source and the destination and are specific to the data packet; hence, these keys are referred to as per-packet keys.

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

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

If the verification of the MAC is not successful, then the ACK packet is silently dropped.

When a NACK packet is received by an intermediate node or the source, the node processes the acknowledgment part of the NACK packet as described above. In addition, it also checks if it has any of the data packets that correspond to the set bits in the bitmap of the NACK packet. If it does not have any of those data packets, it passes on the NACK without modification. Otherwise, for each data packet that it has and that is marked as missing in the NACK packet, it verifies the NACK MAC of the data packet with the corresponding NACK key obtained from the NACK packet. If this verification is successful, then the data packet is scheduled for retransmission, the corresponding bit in the NACK packet is cleared, and the NACK key is removed from the NACK packet. After these modifications, the NACK packet is passed on to the next node towards the source.

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

$$\begin{aligned} K_{ACK} &= \mathrm{PRF}\left(K;\ \text{“}ACK\text{ master key”};\ \mathrm{SessionID}\right), \\ K_{NACK} &= \mathrm{PRF}\left(K;\ \text{“}NACK\text{ master key”};\ \mathrm{SessionID}\right), \end{aligned} \tag{1}$$

where PRF is a pseudorandom function [9] and SessionID is a session identifier.

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

$$K = \mathrm{PRF}\left(S;\ \text{“session master key”};\ \mathrm{SessionID}\right). \tag{2}$$

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

$$\begin{aligned} K_{ACK}^{(n)} &= \mathrm{PRF}\left(K_{ACK};\ \text{“per packet }ACK\text{ key”}; n\right), \\ K_{NACK}^{(n)} &= \mathrm{PRF}\left(K_{NACK};\ \text{“per packet }NACK\text{ key”};\ n\right). \end{aligned} \tag{3}$$

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

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

3. *crypt*: The Calculus for Cryptographic Protocols

crypt is the base calculus for specifying and analysing cryptographic protocols, without supporting real-time and probabilistic systems. *crypt* can be seen as a modified variant of the applied π-calculus [5], designed for analysing security protocols and proving their security properties in a convenient way. Our goal is to extend *crypt* with time and probabilistic modelling elements *adopting the concept of timed and probabilistic automata*, and to do this, we need to modify the applied π-calculus in some points. Namely, we replace process replication with recursive process invocation; we add definition for positive integers and comparison rules for them; we also define syntax for cache/buffer entry.

3.1. Syntax and Semantics. We assume an infinite set of *names* $\mathcal{N}$ and *variables* $\mathcal{V}$, where $\mathcal{N} \cap \mathcal{V} = \emptyset$. Further, we define a set of distinguished variables $\mathcal{E}$ that model the cache entries for entities that store data. In the set $\mathcal{N}$, we distinguish channel names, and other kinds of data. Channel names are denoted by c_i with different indices such that $c_i \neq c_j$, $i \neq j$. The set of nonnegative 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, and so forth.

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

We define a set of terms as

$$t ::= c_i \mid int_i \mid n_i, m_i, k_i \mid x_i, y_i, z_i \mid e_i \mid f\left(t_1, \ldots, t_k\right), \tag{4}$$

where | represents “or.” In particular, a term can be the following.

(i) c_i models a communication channel between honest parties.

(ii) n_i, m_i, and k_i are names and are used to model some data.

(iii) x_i, y_i, and 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 [5].

(iv) e_i is a cache entry.

(v) Finally, f is a function with arity k and is used to construct terms and to model cryptographic primitives and messages. Complex messages are modelled by the function *tuple* with k terms: $tuple(t_1, \ldots, t_k)$, which we abbreviate as $(t_1, \ldots, t_k)$. The function symbol with arity zero is a constant.

For example, digital signature, encryption, and message authentication code (MAC) can be modelled by the functions $sign(m, k)$, $enc(m, k)$, and $MAC(m, k)$, where m models the message to be signed, to be encrypted, and to be MAC-ed, respectively, and k models the secret key. Verification of signatures and MACs, as well as decryption, is modelled by equations. For instance, the equation $dec(enc(m, k), k) = m$ says that, if we decrypt the encryption $enc(m, k)$ with the right key, then we will get the message as a result. If incorrect key is used for decryption then the process will get stuck and will not continue to run.

(vi) int_i ranges over special functions for modelling nonnegative integers. Formally, let 0 be the base element of set $\mathcal{F}$ and is formally defined as the function named by 0. Each further integer is defined as a constructor function named by 1, 2, and so forth. Let the function $inc(int_i)$ be the function that increases the integer int_i by one. Numbers $1, 2, \ldots$, are modelled by functions $inc(0)$, $inc(1)$,..., respectively. The relation between these integers 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:

$$
\begin{aligned}
&P, Q, R ::= \textit{Processes} \\
&\quad \overline{c}\,\langle t\rangle \cdot P \mid c\,(x) \cdot P \mid (P \mid Q) \mid P\,[\,]\,Q \mid \nu n \cdot P \mid \\
&\quad I\,(y_1, \ldots, y_n) \mid \\
&\quad \left[t_i = t_j\right] P \text{ else } Q \mid \\
&\quad \left[int_i \geq int_j\right] P \text{ else } Q \mid \\
&\quad \left[int_i > int_j\right] P \text{ else } Q \mid \left[t_i = t_j\right] P \mid \left[int_i \geq int_j\right] P \\
&\quad \mid \left[int_i > int_j\right] P \mid \mathbf{nil} \mid \text{let } (x = t) \text{ in } P, \\
&\quad \text{let } (e = t) \text{ in } P.
\end{aligned}
\tag{5}
$$

(i) The process $\overline{c}\langle t\rangle \cdot P$ represents the sending of message t on channel c, followed by the execution of P. Process $c(x) \cdot P$ represents the receiving of some message, which is bound to x in P.

(ii) In the composition $P \mid Q$, processes P and Q run in parallel. Each may interact with the other on channels known to both or with the outside world, independently of the other. For example, the communication between the sending process $\overline{c}\langle t\rangle \cdot P$ and receiving process $c(x) \cdot P$ can be described as the parallel composition $\overline{c}\langle t\rangle \cdot P \mid c(x) \cdot Q$. The communication can be described as a reduction step of the parallel composition, namely, $\overline{c}\langle t\rangle \cdot P \mid c(x) \cdot Q \rightarrow P \mid Q\{t/x\}$. In the resulted process, $\overline{c}\langle t\rangle \cdot P$ proceeds to P (meaning that t has been sent), while $c(x) \cdot Q$ proceeds to $Q\{t/x\}$ (meaning that t has been received and now it can be used in Q for further computations).

(iii) A choice $P\,[\,]\,Q$ can behave as either P or Q depending on the first visible/invisible action of P and Q. If the first action of P is enabled but the first action of Q's is not then P is chosen, and vice versa. In case that both actions are enabled the behavior is the same as a nondeterministic choice.

(iv) A restriction $\nu n \cdot P$ is a process that makes a new, private (restricted) name n and then behaves as P. The scope of n is restricted to P and is available only for the process within its scope. A private channel c restricted to P is defined by $\nu c \cdot P$, which does not allow the attackers to eavesdrop on the channel.

(v) A typical way of specifying infinite behavior is by using parametric recursive definitions, like in the π-calculus [10]. Here $I(y_1, \ldots, y_n)$ is an identifier (or invocation) of arity n. We assume that every such identifier has a unique, possibly recursive, definition $I(x_1, \ldots, x_n) \stackrel{def}{=} P$, where x_i's are pairwise distinct. The intuition is that $I(y_1, \ldots, y_n)$ behaves as P with each x_i replaced by y_i, respectively.

(vi) In processes $[t_i = t_j]P$ *else* Q; $[int_i \geq int_j]P$ *else* Q; and $[int_i > int_j]P$ *else* Q: if $(t_i = t_j)$, $(int_i \geq int_j)$ and $(int_i > int_j)$, respectively, then process P is "activated"; else they behave as Q. When Q is the **nil** process, we simply remove the else-branch from the processes.

(vii) The process **nil** does nothing and is used to model the termination of a process behavior.

(viii) Finally, *let* $(x = t)$ *in* P (or *let* $(e = t)$ *in* P) mean that every occurrence of x (or e) in P is bound to t.

We adopt the notion of *environment*, well-known in process algebra, which is used to model the attacker(s) who can obtain the (publicly) exchanged messages and can modify them. Moreover, we adopt the notation of the extended process and active substitution in the applied π-calculus [5] to model the information that the attacker(s) (or the environment) is getting to know during the system run. The definition of the *extended process* is as follows:

$$
A, B, C ::= P \mid (A \mid B) \mid \nu n \cdot A \mid \nu x \cdot A \mid \left\{\frac{t}{x}\right\}. \tag{6}
$$

(i) P is a plain network that we already discussed above.

(ii) $A \mid B$ is a parallel composition of two extended process.

(iii) $\nu n \cdot A$ is a restriction of the name n to A.

(iv) $\nu x \cdot A$ is a restriction of the variable x to A.

(v) $\{t/x\}$ means that the binding of t to x, denoted by $\{t/x\}$, is applied to *any* process that is in parallel composition with $\{t/x\}$. Intuitively, the binding applies to any process that comes into contact with it. To restrict the binding $\{t/x\}$ to a process P, we use the variable restriction νx over $(\{t/x\} \mid P)$, namely, $\nu x \cdot (\{t/x\} \mid P)$. Using this, the equivalent definition of process $\overline{c}\langle t\rangle \cdot P$ can be given by $\nu x \cdot (\overline{c}\langle x\rangle \cdot P \mid \{t/x\})$. Active substitutions are always assumed to be cycle-free.

We define $fv(A)$, $bv(A)$, $fn(A)$, 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:

$$fv\left(\left\{\frac{t}{x}\right\}\right) \stackrel{def}{=} fv(t) \cup \{x\}, \qquad fn\left(\left\{\frac{t}{x}\right\}\right) \stackrel{def}{=} fn(t).$$
$$bv\left(\left\{\frac{t}{x}\right\}\right) \stackrel{def}{=} \emptyset, \qquad bn\left(\left\{\frac{t}{x}\right\}\right) \stackrel{def}{=} bn(t). \tag{7}$$

The concept of bound and free values is similar to local and global scope in programming languages. The scope of names and variables is 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 the 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 restricted names. The set of free variables $fv(A)$ contains the variables x occurring in A which are not restricted variables (νx) or input variable ($c(x)$). A plain process P is closed if it contains no free variable. An extended process is closed when every variable x is either bound or defined by an active substitution.

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

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

3.1.1. Labeled Transition System ($\stackrel{\alpha}{\rightarrow}$). The operational semantics for processes is defined as a labeled transition system $(\mathscr{P}, \mathscr{G}, \rightarrow)$ where $\mathscr{P}$ represents a set of extended processes, $\mathscr{G}$ is a set of labels, and $\rightarrow \subseteq \mathscr{P} \times \mathscr{G} \times \mathscr{P}$.

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

Silent Transition (Internal Computations/Verification) Rules for Processes

(*Let*) let $x = t$ in $P \stackrel{\tau}{\rightarrow} P\{t/x\}$; let $e = t$ in $P \stackrel{\tau}{\rightarrow} P\{t/e\}$.

(*If1*) $[t_i = t_j]P$ else $Q \stackrel{\tau}{\rightarrow} P$; $[t_i = t_j]P \stackrel{\tau}{\rightarrow} P$ (if $t_i = t_j$).

(*If2*) $[t_i = t_j]P$ else $Q \stackrel{\tau}{\rightarrow} Q$; $[t_i = t_j]P \stackrel{\tau}{\rightarrow}$ **nil** (if $t_i \neq t_j$).

(*If3*) $[int_i > int_j]P$ else $Q \stackrel{\tau}{\rightarrow} P$; $[int_i > int_j]P \stackrel{\tau}{\rightarrow} P$ (if $int_i > int_j$).

(*If4*) $[int_i > int_j]P$ else $Q \stackrel{\tau}{\rightarrow} Q$; $[int_i > int_j]P \stackrel{\tau}{\rightarrow}$ **nil** (if $int_i \leq int_j$).

Action Transition (Message Input/Output) Rules for Processes

(In) $c(x) \cdot P \stackrel{c(t)}{\longrightarrow} P\{t/x\}$;

(Open) if $A \stackrel{\overline{c}\langle u\rangle}{\longrightarrow} A', u \neq c$ then $\nu u \cdot A \stackrel{\nu u.\overline{c}\langle u\rangle}{\longrightarrow} A'$.

Let binds a variable to a term in a process; rules *If1–If4* check the relation of two terms or integers. Rule (In) says that, when a term t is received, it is bound to every occurrence of x in P. Rule (Open) defines the output of a restricted name. Note that these rules are examples from the full list of rules, which can be found in [11]. From rule (Open) and the fact that $\nu x \cdot (\overline{c}\langle x\rangle \cdot P | \{t/x\})$ is equivalent to $\overline{c}\langle t\rangle \cdot P$, we have two additional output rules

(Out-1) $\overline{c}\langle t\rangle \cdot P \stackrel{\nu x \cdot \overline{c}\langle x\rangle}{\longrightarrow} \{t/x\} \mid P$,

(Out-2) $\nu n \cdot (\overline{c}\langle t\rangle \cdot P) \stackrel{\nu x.\overline{c}\langle x\rangle}{\longrightarrow} \nu n \cdot (\{t/x\} \mid P)$.

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

$$\overline{c}\langle t_1\rangle \cdot \overline{c}\langle t_2\rangle \cdot P \stackrel{\nu x_1 \cdot \overline{c}\langle x_1\rangle}{\longrightarrow} \left\{\frac{t_1}{x_1}\right\} \mid \overline{c}\langle t_2\rangle \cdot P \stackrel{\nu x_2 \cdot \overline{c}\langle x_2\rangle}{\longrightarrow} \left\{\frac{t_1}{x_1}\right\} \mid \left\{\frac{t_2}{x_2}\right\} \mid P. \tag{8}$$

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

Similarly as in [5], we define an *equational theory Eq*, that is, a set of equations of the form $t_1 = t_2$. This allows us to define cryptographic primitives and operations, such as one-way hash function and MAC computation. For instance, the function Tuple models a tuple of n terms $t_1, t_2, \ldots, t_n$, and its inverse function **i** returns the ith element of a tuple of n elements, where $i \in \{1, \ldots, n\}$. We abbreviate the tuple as $(t_1, t_2, \ldots, t_n)$ in rest of the paper:

$$Tuple(t_1, t_2, \ldots, t_n); \qquad i(t_1, t_2, \ldots, t_n) = t_i. \tag{9}$$

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

$$MAC(t, K); \quad CheckMAC(MAC(t, K), K) = ok \tag{10}$$

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

We can also model many other cryptographic operations and primitives such as symmetric and asymmetric encryption, digital signature, and one-way hash function. However, in this paper we omit to detail them because we do not use these operations. An interested reader can find more details in our longer report [11].

4. $crypt^{prob}_{time}$: Extending *crypt* with Timed and Probabilistic Syntax and Semantics

We propose a time and probabilistic extension to *crypt*, denoted by $crypt^{prob}_{time}$. Our calculus is tailored for the verification of security protocols, especially for verifying protocols that need to cache data, such as transport protocols for wireless sensor networks. This is a new probabilistic timed calculus for cryptographic protocols and, to the best of our knowledge, the first of its kind. The design methodology of $crypt^{prob}_{time}$ is based on the terminology proposed in previous works, it can be seen as the modification and extension of them and contains some novelties.

Namely, the timed extension of *crypt* is based on the timed calculus proposed in [7, 12], and it is also based on the syntax and semantics of the well-known timed automata. The probabilistic extension is inspired by the syntax and semantics of the probabilistic extension of the applied π-calculus proposed in [6] and the probabilistic automata in [7]. The main difference between our work and the related methods is that we focus on extending *crypt*, which is different from the calculus used in those works. In addition, we combine both timed and probabilistic elements at the same time. Finally, we also propose a new definition called *weak probabilistic timed bisimilarity* for proving the existence of the attacks against security protocols.

The concept of $crypt^{prob}_{time}$ is based on the concept of probabilistic timed automata; hence, the correctness of $crypt^{prob}_{time}$ comes from the correctness of the automata because the semantics of $crypt^{prob}_{time}$ is equivalent to the semantics of the probabilistic timed automata, and we show that each process in $crypt^{prob}_{time}$ has an associated probabilistic timed automaton.

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

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

$$\phi ::= true \mid false \mid x_c \sim N \mid \phi_1 \wedge \phi_2 \mid \neg\phi, \tag{11}$$

where ϕ ranges over $\Phi(C)$, $x_c \in \mathscr{C}$, N is a real number, and $\sim \in \{<, \leq, \geq, >\}$. We write $\nu \vDash \phi$ when the valuation ν satisfies the constraint ϕ. Formally, $\nu \vDash true$; $\nu \vDash x_c \sim N$ if and only if $\nu(x_c) \sim N$; $\nu \vDash \phi_1 \wedge \phi_2$ if and only if $\nu \vDash \phi_1 \wedge \nu \vDash \phi_2$.

4.1. Syntax. In the following, we turn to define probabilistic timed processes for $crypt^{prob}_{time}$:

$$\begin{aligned} A_{pt} ::= {} & A \mid \alpha^* \prec_\pi A_{pt} \mid \phi \hookrightarrow A_{pt} \mid \|C_R\| A_{pt} \mid \\ & A^1_{pt} \; [\,] \; A^2_{pt} \mid A^1_{pt} \oplus_p A^2_{pt} \mid \left(A^1_{pt} \mid A^2_{pt}\right) \mid X_{pt}. \end{aligned} \tag{12}$$

We will discuss the meaning of $crypt^{prob}_{time}$ processes by showing the connection between the modeling elements of a probabilistic timed automata and $crypt^{prob}_{time}$. For this purpose, we recall the definition of probabilistic timed automaton [12]: a probabilistic timed automaton *Aut* is defined by the tuple $(\mathscr{L}, l_0, \sum, \mathscr{C}, \mathscr{I}nv, \kappa, E, \Pi)$, where

(i) $\mathscr{L}$ is a finite set of locations and l_0 is the initial location;

(ii) $\sum$ is a set of actions that range over *act*;

(iii) $\mathscr{C}$ is a finite set of clocks;

(iv) $\mathscr{I}nv$: $\mathscr{L} \mapsto \Phi(\mathscr{C})$ is a function that assigns location to a formula, called a location invariant, that must hold at a given location;

(v) κ: $\mathscr{L} \mapsto 2^{\mathscr{C}}$ is the set of clock resets to be performed at the given locations;

(iv) $E \subseteq \mathscr{L} \times \sum \times \Phi(\mathscr{C}) \times \mathscr{B} \times \mathscr{L}$ is the set of edges; we write $l \xrightarrow{act,\phi} l'$ when $(l, act, \phi, B, l') \in E$, where act, ϕ are the action and the time constraint defined on the edge and B is the set of the clocks to be reset at l';

(iiv) $\Pi = \{\pi_1, \ldots, \pi_n\}$ is a finite set of probability distributions; each π_i is a function $\pi_i : E \mapsto [0, 1]$ for

any $i = \{1, \ldots, n\}$, where $\pi_i(g)$ is the probability of an edge g according to distribution π_i and the sum of the edges from a given location l is 1.

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

(i) A_{pt} can be an extended process A without any time construct;

(ii) $\alpha^* \prec_\pi A_{pt}$ performs α^* as the first (not timed) action with the distribution π, at any time, and then it behaves like A_{pt}; note that α^* can be $\nu x.\bar{c}\langle x\rangle$, $\bar{c}\langle u\rangle$, $c(t)$, and the silent action τ; for instance, if A_{pt} is $c(t).P$, where P is the plain process in *crypt*, then α^* is $c(t)$; this process corresponds to the automaton edge $l \xrightarrow{\alpha^*, true}_\pi l'$, where $\alpha^* \prec_\pi A_{pt}$ and A_{pt} corresponds to locations l and l', respectively;

(iii) $\phi \hookrightarrow A_{pt}$ represents a time guard of an action and says that the first action α^* of A_{pt} is performed if the guard (time constraint) ϕ holds; this process intends to model the edge $l \xrightarrow{\alpha^*, \phi}_\pi l'$ in the automaton syntax, where A_{pt} corresponds to l, while the explicit appearance of the target location l' is omitted in the process; in the transition, the action α^* is performed according to the distribution π; when an action has a time guard *true* it means that the action can be performed at any time;

(iv) $\phi \rhd A_{pt}$ represents a clock invariant over A_{pt}; this process corresponds to the location invariant in an automaton; like in timed automaton, this means that the system cannot "stay" in process A_{pt} once the time constraint ϕ becomes invalid; if it cannot move from this process via any transition, then it is a deadlock situation; invariants can be used to model timeout;

(v) in the timed process $\|C_R\|A_{pt}$, first, the clocks in the set C_R are reset and then it behaves like A_{pt} with the reset clock values;

(vi) $A^1_{pt}[\,]A^2_{pt}$ and $A^1_{pt} \mid A^2_{pt}$ describe the first-action choice and the parallel composition of two processes, respectively; process $A^1_{pt} \oplus_p A^2_{pt}$ behaves like A^1_{pt} with probability p, and it behaves as A^2_{pt} with $(1 - p)$; $A^1_{pt}\,[\,]\,A^2_{pt}$ corresponds to a location l from which two edges start, and they are chosen based on the first enable action of A^1_{pt} and A^2_{pt}; for parallel composition, we define $A^1_{pt} \mid A^2_{pt}$ as a location, instead of the parallel composition of two automata; process $A^1_{pt} \oplus_p A^2_{pt}$ corresponds to a location l from which two edges start: $l \xrightarrow{\alpha^*, \phi}_p l_1$ and $l \xrightarrow{\alpha^*, \phi}_{1-p} l_2$, where l_1 and l_2 correspond to A^1_{pt} and A^2_{pt}, respectively;

(vii) X_{pt} is a process variable to which one of the timed processes $\phi \hookrightarrow A_{pt}$, $\phi \rhd A_{pt}$, and $\|C_R\|A_{pt}$ can be bound; Note that this differs from [10], as for our problem, we restrict process variables (X_{pt}) 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 an infinite invocation cycle (e.g., $A = \{t/x\}|A$, where the process variable is bound to A); hence it is not well-defined.

Definition 1. We extend the definition of free and bound variables in Section 3 with the set of clock variables. The set of free variables and bound variables of A_{pt}, denoted by $fvc(A_{pt})$ and $bvc(A_{pt})$, respectively, are as follows.

(i) $fvc(\phi \hookrightarrow A_{pt}) = \text{clock}(\phi) \cup fvc(A_{pt})$: edge guards contains free clock variables.

(ii) $fvc(\phi \rhd A_{pt}) = \text{clock}(\phi) \cup fvc(A_{pt})$: invariant contains free clock variables.

(iii) $bvc(\|C_R\|A_{pt}) = bvc(A_{pt}) \cup C_R$: clocks to be reset are bound clock variables.

(iv) $fvc(A^1_{pt}\,[\,]\,A^2_{pt}) = fvc(A^1_{pt}) \cup fvc(A^2_{pt})$; $bvc(A^1_{pt}\,[\,]\,A^2_{pt}) = bvc(A^1_{pt}) \cup bvc(A^2_{pt})$.

(v) $fvc(A^1_{pt} \mid A^2_{pt}) = fvc(A^1_{pt}) \cup fvc(A^2_{pt})$; $bvc(A^1_{pt} \mid A^2_{pt}) = bvc(A^1_{pt}) \cup bvc(A^2_{pt})$.

(iv) $fvc(A^1_{pt} \oplus_p A^2_{pt}) = fvc(A^1_{pt}) \cup fvc(A^2_{pt})$; $bvc(A^1_{pt} \oplus_p A^2_{pt}) = bvc(A^1_{pt}) \cup bvc(A^2_{pt})$.

The free and bound clock variables of choices and parallel composition are the union of the free and bound clock variables of each process. The reason that the set of clock variables is divided into bound and free parts is to avoid conflicts of clock valuations. For instance, let us consider the process $x_c \le 8 \rhd (\|x_c\|A_{pt})$, in which the clock x_c is reset, and this affects the invariant $x_c \le 8$. Further, in the parallel composition $(\|x_c\|A_{pt})|(x_c \le 8 \rhd A'_{pt})$ the clock variable x_c is the shared variable of the two processes; however, the reset of x_c affects the behavior of process $(x_c \le 8) \rhd A'_{pt}$. This is undesirable since the operational semantics of a process also depends on the behavior of the environment (which is hard to control).

Hence, we define the notion of processes that do not contain any conflict of clock variables, using the following inductive definition and the predicate *ncv* (which refers to "non-conflict of clock variables"):

(1) $ncv(A)$;

(2) $ncv(X_{pt})$;

(3) $ncv(\alpha^* \prec_\pi A_{pt})$ if and only if $ncv(A_{pt})$;

(4) $ncv(\|C_R\|A_{pt})$ if and only if $ncv(A_{pt})$;

(5) $ncv(\phi \hookrightarrow A_{pt})$;

(6) $ncv(\phi \triangleright A_{pt})$: in both cases, if and only if $ncv(A_{pt}) \wedge (clock(\phi) \cap \kappa(A_p t) = \emptyset)$.

Rule 1 holds because an extended process A does not include any clock variables. Rule 2 says that the recursive process invocation of plain processes is nonconflict because a plain process does not contain clock variables. Rule 3 comes from the fact that action α^* is free from clock variables. Rule 4 says that the outermost clock resettings do not cause the conflict of variables in process Apt. Rules 5 and 6 say that if guard and invariant constructs are placed outside then their clock variables cannot be reset within A_{pt}, to avoid conflict. For the full list of *ncv* rules please check our report [11].

In the following, for each $crypt^{prob}_{time}$ process we add rules that associate each process with the invariant and resetting functions ∂ and κ, respectively. Note that we only give the two most important rules in this paper; the full list can be found in [11]. Consider the following:

$$(rk)\ \kappa(\|C_R\|A_{pt}) = C_R \cup \kappa(A_{pt});$$
$$(ri)\ \partial(\phi \triangleright A_{pt}) = \partial(A_{pt}) \wedge \phi.$$

Rule *rk* says that the set of clocks to be reset in $\kappa(\|C_R\|A_{pt})$ is C_R and the clock resets occur in A_{pt}, and rule *ri* says that the invariant of process $\phi \triangleright A_{pt}$ is the intersection of ϕ and the invariant predicate in A_{pt}.

4.2. Operational Semantics. The formal semantics of $crypt^{prob}_{time}$ follows the semantics of probabilistic timed automata. Namely, a state s is defined by the pair (A_{pt}, v), where v is the clock valuation at the location with label A_{pt} with the time issues defined at the location. The initial state s_0 consists of the initial process and initial clock valuation (A^0_{pt}, v_0). Note that the initial process A^0_{pt} is the initial status of a system behavior, while v_0 typically contains the clocks in the reset state. The operational semantics of $crypt^{prob}_{time}$ is defined by a probabilistic timed transition system (PTTS).

A probabilistic timed transition system can be seen as the labeled transition system extended with time and probabilistic constructs. In our model, we follow the concept of [7, 12], but we also improve them with language elements and a new definition of bisimilarity for proving/refuting the security properties of protocols.

Definition 2. Let $\sum$ be the set of actions. A probabilistic timed transition system is defined as the tuple $PTTS = (\mathcal{S}, \sum \times \mathbb{R}^{\geq 0} \times \Pi, s_0, \rightarrow_{PTTS}, \mathcal{U}, F)$, where

(i) $\mathcal{S}$ is a set of states, and s_0 is an initial state;

(ii) $\rightarrow_{PTTS} \subseteq \mathcal{S} \times (\sum \times \mathbb{R}^{\geq 0}) \times \Pi \times \mathcal{S}$ is the set of probabilistic timed labeled transitions; a transition is defined between the source and target state, and the label of the transition is composed of the actions, the time stamp (duration), and the probability of the action; when $(\alpha^*, d, \pi) \in \sum \times \mathbb{R}^{\geq 0} \times \Pi$ we denote the transition from s to s' by $s \xrightarrow{\alpha^*(d),\pi}_{PTTS} s'$; the appearance of π on the arrow means that the transition is performed with the probability according to the distribution π; the label $\alpha^*(d)$ says that performing either a visible α or invisible (silent) τ action ($\alpha^* = \alpha \cup \tau$) consumes d time units; we interpret d as the time for executing action α^*, and there is no idling time at s before performing an action;

(iii) $\mathcal{U} \subseteq \mathbb{R}^{\geq 0} \times \mathcal{S}$ 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 $\mathcal{U}_d$;

(iv) the scheduler F chooses nondeterministically the distribution of action transition steps.

The probabilistic timed transition system *PTTS* should satisfy the two axioms, Until and Delay (in both cases $\Rightarrow$ denotes logical implication):

$$(Until)\ \text{for all } d, d' \in \mathbb{R}^{\geq 0}, \mathcal{U}_d(s) \wedge (d' < d) \Rightarrow \mathcal{U}_{d'}(s),$$

$$(Delay)\ \text{for all } d \in \mathbb{R}^{\geq 0}, s \xrightarrow{\alpha^*(d),\pi}_{PTTS} s' \text{ for some } s' \Rightarrow \mathcal{U}_d(s).$$

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

In addition, $\mathcal{U}$ are the smallest set satisfying the following rules:

$$(u1)\ \mathcal{U}_d(A, v);$$
$$(u2)\ \mathcal{U}_d(\alpha^* \prec_\pi A_{pt}, v);$$
$$(u3)\ \mathcal{U}_d(\phi \hookrightarrow A_{pt}, v) \text{ if } \mathcal{U}_d(A_{pt}, v);$$
$$(u4)\ \mathcal{U}_d(\|C_R\|A_{pt}, v) \text{ if } \mathcal{U}_d(A_{pt}, v[rst : C_R]);$$
$$(u5)\ \mathcal{U}_d(\phi \triangleright A_{pt}, v) \text{ if } \mathcal{U}_d(A_{pt}, v) \wedge \vDash (v + d)(\phi);$$
$$(u6)\ \mathcal{U}_d(A^1_{pt}\ [\ ]\ A^2_{pt}, v) \text{ if } \mathcal{U}_d(A^1_{pt}, v) \vee \mathcal{U}_d(A^2_{pt}, v);$$
$$(u7)\ \mathcal{U}_d(A^1_{pt} \mid A^2_{pt}, v) \text{ if } \mathcal{U}_d(A^1_{pt}, v) \vee \mathcal{U}_d(A^2_{pt}, v);$$
$$(u8)\ \mathcal{U}_d(A^1_{pt} \oplus_p A^2_{pt}, v) \text{ if } \mathcal{U}_d(A^1_{pt}, v) \vee \mathcal{U}_d(A^2_{pt}, v);$$
$$(u9)\ \mathcal{U}_d(X_{pt}, v) \text{ if } \mathcal{U}_d(P[P/X_{pt}], v).$$

Rules (*u1-u2*) are the Until axioms for the states (A, v) and $(\alpha^* \prec_\pi A_{pt}, v)$. In *u3* the system stays in the state $(\phi \hookrightarrow A_{pt}, v)$ until d time units if this is valid to the state (A_{pt}, v) as well. Rules (*u4-u5*) come from the definition of the clock reset and invariant. In rule (*u4*) $v[rst : C_R]$ represents the clock valuation v where the clocks in C_R are reset. Rules (*u6–u8*) say that the system stays until d time units at the state with $A^1_{pt}[]A^2_{pt}$, $A^1_{pt}|A^2_{pt}$, and $A^1_{pt} \oplus_p A^2_{pt}$ if it stays d time in the state with one of the two processes A^1_{pt} and A^2_{pt}. Rule *u9* is concerned with the until predicate for (recursive) process

variable X_{pt}, which comes directly from the definition of recursive process invocation. Note that P is a plain process defined in *crypt*.

We define the satisfaction predicate $\models, \models \subseteq \Phi(\mathscr{C})$, on clock constraints. For each $\phi \in \Phi(\mathscr{C})$ we use the shorthand $\models \nu(\phi)$ if and only if ν satisfies ϕ, for all valuation ν. The set of past closed constraint, $\overline{\Phi(\mathscr{C})} \subseteq \Phi(\mathscr{C})$, is used for defining semantics of location invariant, for all $\nu \in \mathscr{V}$, $d \in \mathbb{R}^{\geq 0}$: $\models (\nu + d)(\phi) \Rightarrow \models (\nu)(\phi)$. Intuitively, this says that if the valuation $\nu + d$, which is defined as $\nu(x_c) + d$ for all clocks x_c, satisfies the constraint ϕ then so does ν. We adopt the variant of time automata used in [7], where location invariant and clock resets are defined as functions ∂ and κ assigning a set of clock constraints $\overline{\Phi(\mathscr{C})}$ and a set of clocks to be reset $\mathscr{R}(\mathscr{C})$, respectively, to a $crypt^{prob}_{time}$ process.

The probabilistic timed transition (action) rules for $crypt^{prob}_{time}$ are given as follows. We provide the connection of each PTTS transition with the edge syntax well-known in probabilistic timed automata. Consider the following:

$(a1)$ $(\alpha^* \prec_\pi A_{pt}, \nu) \xrightarrow{\alpha^*(d),\pi} {}_{PTTS}(A_{pt}, \nu + d)$ if $\alpha^* \prec_\pi A_{pt} \xrightarrow{\alpha^*,true}{}_\pi A_{pt}$;

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

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

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

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

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

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

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

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

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

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

Theorem 3. *For all $crypt^{prob}_{time}$ process A_{pt} and for all closed valuation ν_0, PTTS(A_{pt}, ν_0, F) is indeed the probabilistic timed transition system defined in probabilistic timed automata.*

We defined rules for renaming the clock variables and we showed that the *ncv* property is preserved by clock renaming; hence, the restriction we made to process without conflict of clock variables is harmless [7]. Based on the rules of renaming we also added new rules for *structural equivalent resulted from renaming*. We omit the discussion of these rules in detail because we do not use them, but the reader can find it in our longer report [11].

Weak Probabilistic Timed (Prob-Timed) Labeled Bisimulation. We provide a novel bisimilarity definition, called *weak prob-timed labeled bisimulation* for $crypt^{prob}_{time}$, which enables us to prove or refute the security of probabilistic timed systems.

Our proposed definition makes use of the definition of static equivalence proposed in the applied π-calculus [5], which says that the outputs of static equivalent processes cannot be distinguished by the environment (or attackers). The main advantage of static equivalence is that it only takes into account the static knowledge exposed by two processes to show the behavioral equivalence of them. This method is much easier to use than using the well-known *observation equivalence* [5], where we have to consider the dynamic behavior of processes.

Let the extended process A be $\{t_1/x_1\}|\cdots|\{t_n/x_n\} \mid P_1 \mid \cdots \mid P_n$. The *frame* φ of A is the parallel composition $\{t_1/x_1\} \mid \cdots \mid \{t_n/x_n\}$ that models all the information that is output so far by the process A, which are $t_1, \ldots, t_n$ in this case.

Definition 4 (*static equivalence for extended processes* ($\approx_s$)). 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 include the same number of active substitutions and the 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.

In our proposed *weak prob-timed labeled bisimulation*, we extend the static equivalence with time and probabilistic elements. The meaning of *weak* is that in this paper we want to examine whether the attackers can distinguish the behavior of two processes, based on the information they can *observe*. Hence, in weak prob-timed labeled bisimulation, we do not require the equivalence of the probability of two action traces, because practically an observer cannot distinguish if an action is performed with 1/2 or 1/3 probability.

Nevertheless, we also proposed the definition of *strong prob-timed labeled bisimulation* in our longer report [11], which we do not discuss in this paper, because we found that, for analysing the security of DTSN and SDTP, it is sufficient to use the weak prob-timed labeled bisimulation. *Strong prob-timed labeled bisimulation* is stricter, since it also distinguishes two processes based on the probability of their corresponding action traces.

Definition 5 (*weak prob-timed labeled bisimulation for* $crypt^{prob}_{time}$ *processes*). Let $PTTS_i(A^i_{pt}, v_0, F) = (\mathcal{S}_i, \alpha \times \mathbb{R}^{\geq 0} \times \Pi, s^i_0, \rightarrow_{PTTSi}, \mathcal{U}^i, F)$, and let $i \in \{1, 2\}$ be two probabilistic timed transition systems for $crypt^{prob}_{time}$ processes. Weak prob-timed labeled bisimilarity ($\approx_{pt}$) is the largest symmetric relation $\mathcal{R}$, $\mathcal{R} \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ with $s^1_0 \mathcal{R} s^2_0$, where each s^i is the pair of a closed $crypt^{prob}_{time}$ process and the *same* initial valuation $v_0 \in \mathcal{V}^c$, (A^i_{pt}, v_0), such that $s_1 \mathcal{R} s_2$ implies that

(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 $s_2 \xRightarrow{\tau(\sum d_i),\pi_i}_{PTTS2} s'_2$ for the same F, with $d = f(\sum d_i)$ for some function f, and $s'_1 \mathcal{R} s'_2$;

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

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

The arrow $\overset{\alpha}{\Rightarrow}_{PTTS}$ is the shorthand of the action trace $\xrightarrow{\tau}{}^*_{PTTS} \xrightarrow{\alpha}_{PTTS} \xrightarrow{\tau}{}^*_{PTTS}$, where $\xrightarrow{\tau}{}^*_{PTTS}$ represents a series (formally, a transitive closure) of sequential transitions $\xrightarrow{\tau}_{PTTS}$. $\sum d_i$ on $\Rightarrow_{PTTS}$ is the sum of the time elapsed at each transition and represents the total time elapsed during the sequence of transitions. Note that $fn(A^2_{pt})$ and $dom(A^1_{pt})$ are the same as $fn(A^2)$ and $dom(A^1)$, respectively. Moreover, a process A_{pt} is closed if its nontimed and "nonprobabilistic" counterpart A is closed.

Intuitively, in case that A^1_{pt} and A^2_{pt} 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 based on 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 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 computation) steps. Finally, the third point says that the behavior of the two protocols matches in transition with the action α.

5. Security Analysis of DTSN and SDTP Using $crypt^{prob}_{time}$

We formally prove the insecurity of the DTSN and SDTP protocols using the weak prob-timed bisimilarity defined in $crypt^{prob}_{time}$. We also specified the behavior of the two protocols using the syntax of $crypt^{prob}_{time}$; however, since these descriptions are a bit complicated, for shortening and making the paper readable, we only discuss the most important parts. For the detailed descriptions, the reader is referred to our longer research report [11].

We assume the network topology *S*-*I*-*D*, where "-" represents a bidirectional link, while *S*, *I*, and *D* denote the source, an intermediate node, and the destination node, respectively. We also include the presence of the application that uses DTSN and SDTP, because it sends packet delivery requests to the source and it receives delivered packets. In the rest of the paper we refer to the application as the *upper layer*. Note that the attack scenarios which can be found and proved in this topology are also valid in other topologies including more intermediate nodes. Moreover, we assume that each node has three cache entries, denoted by e^s_k, e^i_k, and e^d_k, $1 \leq k \leq 3$. For brevity we let e^s_{1-3} range over e^s from index 1 to index 3, and the same is true for e^i_{1-3} and e^d_{1-3}. We define symmetric channels between the upper layer and the source, c_{sup}; the upper layer and the destination, c_{dup}; the source and intermediate node, c_{si}; the intermediate node and the destination, c_{id}. Moreover, we define additional channels c_{error} and $c_{sessionEND}$ for sending and receiving error and session end signals.

We define $crypt^{prob}_{time}$ processes *upLayer*, *Src*, *Int*, and *Dst* for specifying the behavior of the upper layer, the source, intermediate, and destination nodes. The DTSN protocol for

the given topology is specified by the parallel composition of these four processes:

$$Prot(params) \overset{def}{=}$$
$$\text{let } (e_1^s, e_2^s, e_3^s, e_1^i, e_2^i, e_3^i, e_1^d, e_2^d, e_3^d, cntsq) = (E, E, E, E, E, E, E, E, E, 1) \text{ in } INITDTSN();$$

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

$$DTSN(params) \overset{def}{=}$$
$$upLayer(incr(cntsq)) \mid \{x_c^{act} \le T_{act}\} \triangleright initSrc(\text{s, d, apID}, e_{1-3}^s, \text{sID, earAtmp}) \mid \text{Int}(e_{1-3}^i) \mid Dst(e_{1-3}^d, \text{ ackNbr, nackNbr, toRTX1, nxtsq}).$$

We refer to the tuple of parameters (*cntsq, s, d, apID,* e_{1-3}^i*, sID, earAtmp,* e_{1-3}^i*,* e_{1-3}^d*, ackNbr, nackNbr, toRTX1, nxtsq*) by (*params*). The process *Prot*(*params*) describes DTSN with variable initializations. The *let* construct is used to initialize the value of the cache entries to *E* and the current sequence number to 1. The unique name *E* is used to represent the empty content. In the following, we give a brief overview of the main processes in our specification. Each main process is composed of additional subprocesses, which we skip discussing here. The processes are recursively invoked in a way to model replication. Interested readers can find the full description in [11].

We introduce two clock variables: x_c^{act} for the activity timer and x_c^{ear} for the ear timer. According to the specification of the DTSN protocol [2], to model timeout we make use of the clock invariant defined on the process *Src*. The initial state of DTSN for the given topology is specified as the process $\|x_c^{act}, x_c^{ear}\|Prot(params)$, which simply resets the timers at the beginning. We define the time amount of the activity and ear timers by T_{act} and T_{ear}, respectively. We assume that the activity timer is launched after the upper layer has sent the first request for the source, which is specified in *INITDTSN*().

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

The Source Handling the Activity Timer Expiration

$$InitSrc(\text{s, d, apID}, e_{1-3}^s, \text{sID, earAtmp}) \overset{def}{=}$$
$$c_{sup}(x_{sq}) \cdot (\{x_c^{act} \le T_{act}\} \triangleright initFwdDt(\text{s, apID}, e_{1-3}^s, \text{sID}, x_{sq}) [\,]\{x_c^{act} \le T_{act}\} \triangleright initRcvACKS(\text{s, d, apID}, e_{1-3}^s, \text{sID, earAtmp}) [\,]\{x_c^{act} \le T_{act}\} \triangleright initRcvNACKS(\text{s, d, apID}, e_{1-3}^s, \text{sID, earAtmp}))$$
$$[\,]\{x_c^{act} \le T_{act}\} \triangleright c_{sessionEND}(= SEND) \cdot \mathbf{nil}$$
$$[\,]\{x_c^{act} \le T_{act}\} \triangleright (x_c^{act} = T_{act}) \hookrightarrow \tau.actTimeOut.$$

The process *initSrc* specifies the source node starting with the first packet delivery, when the source does not launch the EAR timer yet but only the ACT (activity) timer. This is defined by the clock invariant construct $\{x_c^{act} \le T_{act}\} \triangleright$ before *initSrc*. The subprocess *actTimeOut* describes the behavior of the protocol when the ACT timer expires. Similarly, for the EAR timer, we define the process *earTimeOut*.

The three choice options represent the "wait for event" behavior of the source. Each choice option represents a scenario. The last (third) option describes the case when the activity timer has elapsed. The format $\{x_c^{act} \le T_{act}\} \triangleright (x_c^{act} = T_{act}) \hookrightarrow \tau.actTimeOut$ follows the concept of the timed automaton and says that when the time has elapsed the system should proceed with the process $\tau.actTimeOut$, which describes the defined behavior of *Src* after timeout. τ is a silent step, moving silently to *actTimeOut*. In this process, we use $\tau.actTimeOut$ instead of *actTimeOut*, because we insist on the semantics of timed automaton. The second choice is for the case when the session is terminated, which happens when the constant *SEND* has been sent on the private channel $c_{sessionEnd}$. The session end signal (i.e., the constant *SEND*) is sent on $c_{sessionEnd}$ by *Src* according to [2]. We assume that the session termination cannot be interrupted by the timeouts; basically, it can be seen as an atomic action. When the first option has been chosen, it means that in that step the source received the delivery request from the upper layer and that no session end or timeouts happen during this input action. The choices among the possible actions are repeatedly put into each subprocess *initFwdDt*, *initRcvACKS*, and *initRcvNACKS*. Namely, the *session end*, *activity*, and/or *EAR timeouts* options are placed before the action steps to be performed.

Process That Models the Behavior of an Intermediate Node

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

In process *Int* the intermediate node may receive a data packet on channel c_{si}, in which case it handles the received packet according to the definition of DTSN, it can receive and handle an ACK or NACK message, and it can terminate its operation when it gets the session end signal (i.e., the constant *SEND*).

We also add a probabilistic choice in the specification. 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 node I. 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 subprocess *hndleDtI*, which is responsible for handling a received data packet. Consider the following

$$strAndFwI\left(\text{s, d, apID, sID, sq, ear, rtx}, e^i_{1-3}\right) \oplus_p FwI\left(\text{s, d, apID, sID, sq, ear, rtx}, e^i_{1-3}\right). \tag{13}$$

Process *strAndFwI*, which describes the case when the intermediate node stores (and forwards) the received packet, is chosen with probability p, and process *FwI* that specifies the only forwarding case is selected with probability $1 - p$.

Process That Models the Behavior of the Destination:

$$Dst(e^d_{1-3}, \text{ackNbr, nackNbr, toRTX1, nxtsq}) \stackrel{def}{=} c_{id}((x_s, x_d, x_{apID}, x_{sID}, x_{sq}, x_{ear}, x_{rtx})) \cdot hndleDtDst \; [\,]c_{sessionEND}(= SEND) \cdot \mathbf{nil}.$$

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

To model the cryptographic primitives and operations in SDTP, we add the following equations into the set of equational theories:

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

Equations: CheckMac*(MAC(t, K(n, ACK)), K(n, ACK)) = ok.*

CheckMac*(MAC(t, K(n, NACK)), K(n, NACK)) = ok,*

where functions $K(n, ACK)$ and $K(n, NACK)$ specify the *ACK* and *NACK* per-packet keys corresponding to the packet with sequence number n. In order to simplify the modelling procedure, without violating the correctness of SDTP, we make an abstraction of the key hierarchy given in [3], where the per-packet keys are computed with a one-way function based on the shared secret unknown to the attacker. Instead, we assume that $K(n, ACK)$ and $K(n, NACK)$ cannot be generated (but can be intercepted) by the attackers. The attackers can only generate keys that differ from these keys. With this, we model the fact that the shared secret will never be revealed during the protocol.

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 and then sends it to node I, which is accomplished by the following code part in the processes *initSrc* and *Src*.

The Source Sends a Data Packet in SDTP

let (*ear, rtx, earAtmp*) = (*val1, val2, val3*) in

let (*Kack, Knack*) = (*K(sq, ACK), K*(*sq, NACK*)) in

let *ACKMAC* = *MAC*((*s, d, apID, sID, sq, ear, rtx*), *Kack*) in

let *NACKMAC* = *MAC*((*s, d, apID, sID, sq, ear, rtx*), *Knack*) in

$\overline{c_{si}}\langle$(*s, d, apID, sID, sq, ear, rtx, ACKMAC, NACK-MAC*)$\rangle$.

In the first row, the variables *ear*, *rtx*, and *earAtmp* are given some values *val1*, *val2*, and *val3*, respectively. In the second row the ACK/NACK keys *Kack* and *Knack* are generated, while in the third and fourth rows the ACK/NACK MACs are computed using the generated ACK/NACK keys. Finally, the following code part in *Src* (*initSrc*) models the case when the EAR (ACT) timer is reset (launched) and the source gets back to the idling state. In the two processes (PR1-2), the clock is reset and then the timeout conditions are defined based on clock invariants (in the form $\{x^{act}_c \le T_{act}, x^{ear}_c \le T_{ear}\}$).

Process PR1: Reseting (Launching) the EAR Timer. Consider the following:

$\|x^{ear}_c\|\{x^{act}_c \le T_{act}, x^{ear}_c \le T_{ear}\} \triangleright Src$(s, d, apID, e^s_{1-3}, sID, earAtmp).

Process PR2: Reseting (Launching) Both the EAR and ACT Timers. Consider the following:

$\|x^{ear}_c, x^{act}_c\|\{x^{act}_c \le T_{act}, x^{ear}_c \le T_{ear}\} \triangleright Src$(s, d, apID, e^s_{1-3}, sID, earAtmp).

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

Definition 6. Let the processes *Prot*() and $Prot^{ideal}$() specify the real and ideal versions of some protocol *Prot*, respectively. We say that *Prot* is secure (up to the strictness of the ideal version) if *Prot*() and $Prot^{ideal}$() are probabilistic timed bisimilar: $Prot() \approx_{pt} Prot^{ideal}()$.

The strictness of the security requirement, which we expect a protocol to fulfill, depends on how ideally/securely we specify the ideal version. Intuitively, Definition 6 says that *Prot* is secure if the attackers, who can observe the output

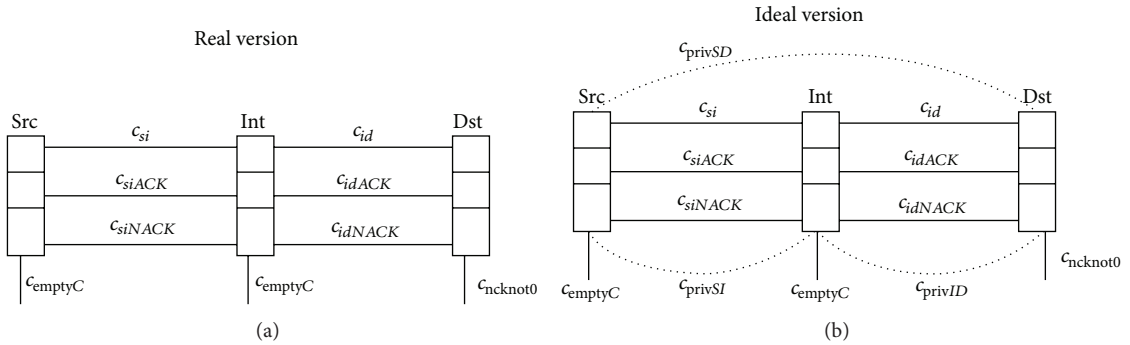

Figure 1: The difference between the real and ideal version of the DTSN and the SDTP protocols.

messages on public channels, cannot distinguish the operation of the two instances.

The main difference between the ideal and the real systems is that, in the ideal system, honest nodes are always 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 1 we show the difference in more detail. In the ideal case, three private channels are defined which are not available to the attacker(s). *Src*, *Int,* and *Dst* denote the processes for the source, the intermediate, and the destination nodes. Channels c_{privSD}, c_{privID}, and c_{privSI} are defined between processes *Src* and *Dst*, *Int* and *Dst*, and *Src* and *Int*, respectively. In the rest of the paper, we refer to the source, intermediate, and destination nodes as *S*, *I*, and *D*. Whenever *S* sends a packet *pck* on public channel c_{si}, it also informs *I* about what should *I* receive, by sending at the same time *pck* directly via private channel c_{privSI} to *I*, so when *I* receives a packet via c_{si} it compares the message with *pck*. The same happens when *I* sends a packet to *D*. Whenever, an honest node receives an unexpected data, it interrupts its normal operation. The channels c_{privSD} and c_{privID} can be used by the destination to inform *S* and *I* about the messages to be retransmitted. We recall that the communication via a private channel is not observable by the environment; hence, it can be seen as a silent τ transition. Note that, for simplicity, we omitted to include the upper layer and channel c_{sup} in the figure, but we put them in our specification. Finally, we also add additional public channels c_{emptyC} and $c_{ncknot0}$ for signalling that the cache has been emptied and that the number of packets to be retransmitted is larger than 0, respectively. These additional channels are defined only for applying bisimilarities in the security proofs, but they do not affect the correctness of the protocol.

With this definition 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. Namely, this means that the honest nodes only handle the messages received on public channels when they are equal to the expected messages received on private channels.

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

To describe the activity of the attacker(s), we apply the concept of the environment, used in the applied π-calculus [5] that models the presence of the attacker(s) in an implicit way. Every message that is output on a public channel is available for the environment; that is, the environment can be seen as a group of attackers who can share information with each other, for instance, via a side channel.

5.1. Security Analysis of the DTSN Protocol. The security properties we want to check in case of the DTSN protocol is that how secure it is against the manipulation of control and data packets. In particular, can the manipulation of packets prevent DTSN from achieving its design goal? In this section we demonstrate how to formally prove the security or vulnerability of DTSN using $crypt^{prob}_{time}$.

First of all, we assume that, in both DTSN and SDTP, each action (verification, sending, and receiving on public channel) takes an equal amount of time d and the function f in Definition 5 returns $\sum d_i$. This assumption does not change the correctness of the protocols. We define the ideal version of the process *Prot(params)*, denoted by $Prot^{ideal}(params)$, which contains the ideal version of *DTSN(params)*:

$$
\begin{aligned}
&/* \textit{ The Ideal Version of the DTSN Protocol for the Given Topology } */\\
&Prot^{ideal}(params) \stackrel{def}{=}\\
&\quad \text{let } (e_1^s, e_2^s, e_3^s, e_1^i, e_2^i, e_3^i, e_1^d, e_2^d, e_3^d, cntsq)\\
&\quad = (E, E, E, E, E, E, E, E, E, 1)\\
&\quad \text{in } INITDTSN^{ideal}(),
\end{aligned}
$$

where process $INITDTSN^{ideal}()$ contains $DTSN^{ideal}(params)$ instead of DTSN (*params*).

To prove or refute the bisimilarity relation, we define *Prot(params)* and $Prot^{ideal}(params)$ such that the source and intermediate nodes output the constants *CacheEmptyS* and *CacheEmptyI* on the public channel c_{emptyC}, respectively, whenever they have emptied their buffers after processing an ACK or a NACK message. This is defined by the following $crypt^{prob}_{time}$ code fragment (for $i \in \{1, 2, 3\}$), where process *checkEi* corresponds to the *i*th cache entry:

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

/* *Cache entry* e^s_i *is emptied and the number of the empty caches is increased* */

let $(e^s_i, nbrEcacheS) = (E, inc\ (nbrEcacheS))$ in

. . . . . .

/* *Here we add the resetting of the two timers on process Src* */

$\{x^{act}_c \leq T_{act}, x^{ear}_c \leq T_{ear}\} \triangleright$ let $(earAtmp = 0)$ in

/* *If the cache has been emptied (emptycacheS = 3) then CacheEmptyS is output* */

[$emptycacheS = 3$]

$\overline{c_{emptyC}}\langle CacheEmptyS\rangle.\ \|x^{act}_c, x^{ear}_c\|\{x^{act}_c \leq T_{act}\} \triangleright$ Src(s, d, apID, e^s_{1-3}, sID, earAtmp)

else $\|x^{act}_c, x^{ear}_c\|\{x^{act}_c \leq T_{act}\} \triangleright Src$(s, d, apID, e^s_{1-3}, sID, earAtmp).

The specification of intermediate node is similar to the case of the source, but *nbrEcacheI* and *CacheEmptyI* are used instead of *nbrEcacheS* and *CacheEmptyS*, respectively. The constants *CacheEmptyS* and *CacheEmptyI* are output whenever the number of the empty cache entries, *emptycacheS*, *emptycacheI,* is 3, which means that the buffers of S and I are emptied, respectively.

Lemma 7. *With the defined attacker model* $\mathcal{M}_A$*, the DTSN protocol is insecure against message manipulation attacks.*

According to Definition 5 processes *Prot(params)* and $Prot^{ideal}(params)$ are not weak prob-timed bisimilar because each point of the definition is violated. The following proof show that DTSN is vulnerable to the manipulation of control packets: in SC-1 the attacker increases the base number in ACK packets causing the stored packets to be deleted from the cache although they should not be, while in SC-2 the attacker forces the destination node to send unnecessary ACKs or NACKs.

(i) *Scenario SC-1.* This scenario can happen beside the topology *S-I* that includes the attacker *A* within the transmission range of both *S* and *I*. We show that the trace of probabilistic timed transitions in the real system, denoted by $PTTR_{realSC1}$, cannot be simulated with any corresponding trace in the ideal system. The trace $PTTR_{realSC1}$ describes the scenario where the source sends the first packet (sequence number 1) to the intermediate node on channel c_{si}. Because c_{si} is public, this packet is obtained by the attacker(s) (i.e., the environment), who, instead of forwarding it, sends an ACK with base number 1 to *S*. *S* received this message on c_{siACK}, empties its buffer, and outputs the constant *CacheEmptyS* on the public channel c_{emptyC}. As *CacheEmptyS* will not be output in $Prot^{ideal}$ if node *I* has not sent anything, the first point of Definition 5 is violated.

(ii) *Scenario SC-2.* We prove that DTSN is susceptible to the attacks that cause futile energy consumption, by showing the violation of the second point of Definition 5. This scenario can happen in the topology *S-I-D* that includes the attacker *A* within the transmission range of both *I* and *D*. The following trace $PTTR_{realSC2}$ in the real system cannot be simulated by the ideal system: *I* sends a correct packet towards *D*, which is intercepted by *A*. Then *A* forwards the packet to *D* but setting the EAR bit in it to 1, requiring *D* to send an *ACK* or a *NACK*. Let the series of silent transitions $s \xrightarrow{\tau(d),\pi} {}^*_{PTTS} s'$ describe the verification steps made by *D* after receiving the incorrect packet from *A*. Although at this time there is not any difference in the message outputs, the ideal system still cannot simulate this silent trace, because in this case *D* performs only one comparison step, which takes less time units.

5.2. Security Analysis of the SDTP Protocol. We define the ideal version of process *ProtSDTP(params)*, denoted by $ProtSDTP^{ideal}(params)$, in the same concept as in $Prot^{ideal}(params)$. The only difference is that, in SDTP, the processes *Src* and *Int* are defined such that, whenever the MAC verification made by *S* and *I* on the received ACK/NACK message fails or an unexpected message is received, *S* and *I* output a predefined constant *BadControl* via the public channel c_{badpck}. Note that this extension does not affect the correctness of SDTP and only plays a role in the proofs of weak prob-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.

(i) *Proving SC-1.* We prove that SDTP is not vulnerable to the attack scenario (SC-1) by showing that $ProtSDTP^{ideal}(params)$ can simulate (according to Definition 5) the transition trace produced by *ProtSDTP(params)*. In SDTP the packet sent by *S* includes the ACK MAC and NACK MAC. Hence, when the attacker *A* sends the ACK to *S*, in both the real and the ideal systems, the source node outputs the constant *BadControl* on the channel c_{badpck}, because either the MAC verification fails (in the real system) or the

received packet is not the expected one (ideal system). Recall that the MAC verification fails because the attacker does not possess the ACK/NACK keys of the source.

(ii) *Proving SC-2*. Similarly, SDTP is not vulnerable to the attack scenario (SC-2) either. $ProtSDTP^{ideal}(params)$ can simulate the transition trace produced by *ProtSDTP(params)*. After receiving an incorrect packet with EAR bit set to 1, in both the real and the ideal systems, the destination node outputs the constant *BadControl* on the channel c_{badpck}, because either the MAC verification fails (in the real system) or the received packet is not the expected one (ideal system). Hence, they consume equal time units.

As we can see, with the security extensions SDTP could eliminate the essential weaknesses of DTSN; however, we will prove that it is still vulnerable, by showing a trace in the real system *ProtSDTP(params)*, which cannot be simulated in $ProtSDTP^{ideal}(params)$.

Lemma 8. *The SDTP protocol is insecure besides the attacker model* $\mathscr{M}_A$.

To prove the vulnerability of SDTP using prob-timed bisimilarity, we relax the definition of the ideal version such that the honest nodes only compare the received ACK/NACK messages with the expected ones. When they receive a data packet they proceed in the same way as the real version, namely, without any comparison with the expected message that it receives on private channels.

Proving SC-3. We consider the trace that refers to the topology *S*-*A*1-*I*-*A*2. The trace describes the following scenario: *A*1 has received the first packet from *S*, it replaces the MACs computed by *S* with the MACs it computes on the same data (denoted by $ACKMAC_{att}$, $NACKMAC_{att}$), and it sends the modified packet to *I*. Node *I* forwards the packet to *A*2, which, instead of forwarding it to *D*, sends back the ACK message for this data packet to *I*, including the corresponding ACK key for the MAC $ACKMAC_{att}$. As the result, node *I* deletes its buffer and outputs the constant *CacheEmptyI* on the public channel c_{emptyC}. This transition cannot be simulated by any corresponding transition trace in $ProtSDTP^{ideal}(params)$. Because, in this case, the ACK sent by A2 will be received on $c_{idACK}(x^{rcv})$; then, node *I* interrupts its operation. Hence, *CacheEmptyI* will never be output.

Note that we also performed verification and showed other weaknesses of SDTP; the reader can find them in our technical report [11].

6. Automated Security Verification Using the PAT Process Analysis Toolkit

Related Methods. SPIN model checker [14] and UPPAAL [15] are general purpose model checking tools. CPAL-ES [16] and ProVerif [13] are automatic verification tools developed for verifying security protocols. The main drawback of them is that they lack semantics and syntax for defining the systems that include probabilistic and real-time behavior. Hence, they *cannot be used* to verify WSN transport protocols such as DTSN and SDTP. PRISM model checker [17] supports probabilistic and real-time systems but its limited specification language does not enable us to verify protocols/sytems that may perform complex computations.

Our Method. Our method is based on the PAT process analysis toolkit. PAT [8] is a self-contained framework to specify and automatically verify different properties of concurrent (i.e., supporting parallel compositions construct), real-time systems with probabilistic behavior. It provides a user friendly graphical interface, a featured model editor, and an animated simulator for debugging purposes. PAT implements various state-of-the-art model checking techniques for different properties such as reachability, LTL properties with fairness assumptions, refinement checking, and probabilistic model checking. To handle large state spaces, the framework also includes many well-known model checking optimization methods.

One of the biggest advantages of PAT compared with other solutions is that it supports probabilistic and timed, CSP-like behavioral syntax and semantics, which are important in our case. 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 multilift and pacemaker systems. However, PAT (so far) does not provide syntax and semantics for specifying cryptographic primitives and operations, such as digital signature, MAC, encryptions and decryptions, and one-way hash function. Hence, we model cryptographic operations used by SDTP in an abstract, simplified way. Note that the simplification 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. It provides a CSP [18] 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 communication channels, array of variables and channels, similarly to Promela [19] (Process Meta Language), the specification language used by the SPIN [19] model-checker. PAT handles time in a tricky way; namely, instead of modeling clocks and clock resets in an explicit manner, to make the automatic verification more efficient it applies an implicit representation of time (clocks).

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

Communicating Sequential Programs (CSP#) Module. The CSP# module supports a rich modeling language named CSP# (a modified variant of CSP) that features process algebra operators like (conditional or nondeterministic) choices, interrupt, parallel composition, interleaving, hiding, asynchronous message passing channel, and mathematical operators like summation, multiplication.

It also provides low-level constructs like *arrays*, *if-then-else*, and *while*. The modeling of communication among processes is based on message passing via communication channels. Communication channels sending and receiving on a channel can be defined with the following syntax:

(1) `(declaration of channel channame): channel channame size;`

(2) `(output of the msg tuple (m1,m2,m3) on channame):channame!m1.m2.m3;`

(3) `(input a msg (m1,m2,m3) on the channel channame): channame?x1.x2.x3;`

channel is a keyword for declaring channels only, *channame* is the channel name, and *size* is the channel buffer size. It is important that a channel with buffer size 0 sends/receives messages synchronously. A process is a relevant specification element in PAT that is defined as an equation $P(x1, x2, \ldots, xn) = ProcExp$, where *ProcExp* defines the behavior of process *P*. PAT defines special processes to make coding be more convenient: process *Stop* is the deadlock process that does nothing; process *Skip* terminates immediately and then behaves exactly in the same way as *Stop*.

Events are defined in PAT to 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*. An event *ev* can be a simple event or can be combined with assignments which update global variables as in the following example: $ev\{x = x+1;\}$ -> *Stop*, where *x* is a global variable.

A *sequential composition* of two processes *P* and *Q* is written as *P*; *Q* in which *P* starts first and *Q* starts only when *P* has finished. A (general) choice is written as *P*[ ]*Q*, which states that either *P* or *Q* may be executed. If *P* performs an event first, then *P* takes control. Otherwise, *Q* takes control. Interleaving represents two processes, which run concurrently, and is denoted by *P*|||*Q*.

Real-Time System (RTS) Module. The RTS module in PAT enables us to specify and analyse real-time systems and verify timing properties. To make the automatic verification more efficient, unlike timed automata that define explicit clock variables and capture real-time constraints by explicitly setting/reseting clock variables, PAT defines several timed behavioral patterns to capture high-level quantitative timing requirements, such as *wait*, *timeout*, *deadline*, *waituntil*, *timed interrupt*, and *within*. For instance, process *P interrupt[t] Q* behaves as *P* until *t* time units elapse and then it switches to *Q*.

Probability RTS (PRTS) Module. The PRTS module supports means for analyzing probabilistic real-timed systems by extending the RTS module with probabilistic choices and assertions. The most important extension added by the PRTS module is the probabilistic choice (defined with the keyword *pcase*):

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

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

PAT supports a probabilistic assertion, which is a query about the system's probabilistic behaviors, namely, the reachabiliy of a defined goal with some probability:

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

this returns respectively the probability, the minimal probability, and the maximal probability of the event that process *prtsP*()reaches a state where the boolean expression *cond* is true.

6.1. On Verifying SDTP Using the PAT Process Analysis Toolkit. We verify SDTP assuming the topologies *Top1*: *S*-A_1-*I*-*D*, *Top2*: *S*-*I*-A_2-*D*, and *Top3*: *S*-A_1-*I*-A_2-*D*. Following the concept in Section 5, we define public (symmetric) channels between each node pair and define additional symmetric channels *chASPck*, *chASAck*, *chASNack*, *chASEAR*, *chADPck*, *chADAck*, *chADNack*, and *chADEAR*, between the attacker(s) and its (their) honest neighbors.

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

$$K_{ACK}^{(n)} = PRF\left(K_{ACK}; \text{“per packet } ACK \text{ } key\text{”}; n\right),$$
$$K_{ACK}^{(n)} = \mathrm{PRF}\left(K_{KACK}; \text{“per packet } NACK \text{ key”}; \ n\right). \tag{14}$$

Following this concept, in PAT we define the ACK key and NACK key for the packet with sequence number *n* by the "pair" *n.Kack* and *n.Knack*, respectively. To reduce the verification complexity we made abstraction on the key generation procedure and model the session ACK/NACK master keys by the unique constants *Kack* and *Knack*. Then we specify the packets sent by the source node as follows: *sq.ear.rtx.sq.sq.Kack.sq.sq.Knack*, where the first part *sq.ear.rtx* contains the packet's sequence number and the EAR and RTX bits, respectively; the second part *sq.sq.Kack* and the third part *sq.sq.Knack* represent the ACK MAC and NACK MAC computed over the packet with sequence number *sq* without the EAR and RTX bits, using the per-packet ACK and NACK keys *sq.Kack* and *sq.Knack*. An ACK message has the following forms: *acknbr.acknbr.Kack*, where *acknbr.Kack*

is the corresponding ACK key of *acknbr*. A NACK message has the format *acknbr.nckb1.acknbr.Kack.nckb.Knack*, where *nckb.Knack* is the NACK key of the packet to be retransmitted. The NACK message can include more bits, in a similar way.

By default, the attackers do not possess the two master keys *Kack* and *Knack* of honest nodes but only their own key *Katt*. Because honest nodes are specified to wait for these MACs format, the attackers should compose the MACs in this format as well, namely, *sqA.sqA.Katt*. The attackers cannot use the master keys to construct the per-packet ACK/NACK keys, and when they obtain a MAC, for example, *sq.sq.Kack*, they cannot use *sq.Kack*, only in case they receives *sq.Kack*.

We distinguish the following scenarios and examine the possible ability of the attacker(s). The behaviors of the attackers are defined as the processes *procA1()* and *procA2()*. In our model, by default the attackers have two sequence numbers *seqA1* and *seqA2* which are the smallest (i.e., 1) and the largest possible sequence numbers, respectively. The attackers can include $earA \in \{0, 1\}$, $rtxA \in \{0, 1\}$ in their data packets. The attackers, in addition, possess the predefined values $bA1, \ldots, bA4$ for requiring retransmission in NACK messages.

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

(1) *Without Receiving Any Message.* (i) $A1$ sends a data packet, with *seqA1.earA.rtxA* or *seqA2.earA.rtxA*, to I; (ii) $A1$ sends an ACK, for the packet *seqA1* or *seqA2*, to I or to S; (iii) $A1$ sends a NACK, with the ack numbers *seqA1* or *seqA2*, and a combination of $bA1, \ldots, bA4$, to I or to S.

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

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

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

We denote the attackers with this kind of behavior by $\mathscr{M}_A^{PAT}$. The SDTP protocol with the second topology is specified as the parallel compositions of each honest node and the attacker A1:

$$\text{SDTPA1}() = \text{procS}() \,|||\, \text{procA1}() \,|||\, \text{procI}() \,|||\, \text{procD}(). \tag{15}$$

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

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

$$\begin{aligned}\text{SDTPA2}() = {} & \text{UpLayer}() \,|||\, \text{procS}() \\ & |||\, \text{procI}() \,|||\, \text{procA2}() \,|||\, \text{procD}().\end{aligned} \tag{16}$$

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

$$\begin{aligned}\text{SDTPA1A2}() = {} & \text{UpLayer}() \,|||\, \text{procS}() \,|||\, \\ & \text{procA1}() \,|||\, \text{procI}() \,|||\, \\ & \text{procA2}() \,|||\, \text{procD}().\end{aligned} \tag{17}$$

Assertions. The assertion, denoted by *violategoal1*, for verifying the provision of reliable delivery of packets, is as follows:

#define violategoal1 (*OutBufL* == 0 &&*BufI* == 0 && *num NACK* > 0),

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

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

(B1) *#assert SDTP() reaches violategoal1p1;*

(B2) *#assert SDTP() reaches violategoal1p1 with pmax.*

```
PAT code:
  #define violategoal2 (freenum > acknum)
B5. #assert SDTPsubA1subA2() reaches violategoal2
B6. #assert SDTPA1A2() reaches violategoal2
```

Algorithm 1

```
subA1() =
 /* A1 sends a data packet to I, without receiving any message, OR */
 A1NotRcvSndPck2I()
 /* After receiving a data packet on channel chSAPck */
 [] chSAPck?seq.ear.rtx.seq1.seq2.Kack.seq3.seq4.Knack ->
   (
  /* A1 sends a data packet to I, OR */
    A1RcvPckSndPck2I()
  /* A1 forwards the packet unchanged to I */
    [] chIAPck!seq.ear.rtx.seq1.seq2.Kack.seq3.seq4.Knack -> subA1()
  )

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

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

Algorithm 2

For the topology *Top2*, we run PAT to model-check (B1) and get the result *Not Valid*. This means that, in the presence of the defined attacker(s) $\mathscr{M}_A^{PAT}$, SDTP cannot be corrupted such that D has not received some packets and required retransmissions but the buffers of S and I are emptied. Assertion (B2) also results in *Not valid*.

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

Let the number of buffer entries that are freed at node I after receiving an ACK/NACK message be *freenum*, and let the number of packets received in sequence by node D be *acknum*. The bad state *violategoal2* specifies the state where (*freenum* > *acknum*) (see Algorithm 1).

We run the PAT model checker with the attacker processes (see Algorithm 2).

As a result, the tool returned *Valid* for the assertions $B5$ and $B6$ along with the following trace.

(1) A1 sends to I a data pck *seqA2.ear.rtx* with the corresponding ACK MACs: *seqA2.seqA2.Katt* and NACK MACs: *seqA2.seqA2.Katt*.

(2) I stores this pck and forwards it (unchanged) to A2.

(3) A2 received this packet and sends to I the ACK for seqA2: *seqA2.seqA2.Katt.seqA2.Katt* with the 2 keys *seqA2.Katt* and *seqA2.Katt*.

(4) As result, I deletes all the packets stored in its buffer because the key *seqA2.Katt* and the MAC *seqA2.seqA2.Katt* match.

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

We also performed verification and showed other weaknesses of SDTP; the reader can find them in our technical report [11].

7. Conclusion

In this paper, we addressed the problem of formal and automated security verification of WSN transport protocols that may perform cryptographic operations. The verification of this class of protocols is difficult because they typically consist of complex behavioral characteristics, such as real-time, probabilistic, and cryptographic operations. To solve this problem, we proposed a probabilistic timed calculus for cryptographic protocols and demonstrated how to use

this formal language for proving security or vulnerability of protocols. To the best of our knowledge, this is the first such process calculus that supports an expressive syntax and semantics, real-time, probabilistic, and cryptographic issues at the same time. Hence, it can be used to verify systems that involve these three properties. In addition, we proposed an automatic verification method, based on the PAT process analysis toolkit for this class of protocols. For demonstration purposes, we apply the proposed manual and automatic proof methods for verifying the security of DTSN and SDTP, which are two of the recently proposed WSN transport protocols, and showed that (i) DTSN is vulnerable to packet manipulation attacks, (ii) SDTP successfully patched some security holes found in DTSN, and (iii) SDTP is still vulnerable to the sandwich type attack.

In the future, we focus on improving the automatic security verification for this class of systems/protocols. Currently we found that PAT is the most suitable tool because it enables us to define concurrent, nondeterministic, real-time, and probabilistic behavior of systems in a convenient way. However, in its current form it does not support (or only in a very limited way) cryptographic primitives and operations, as well as the behavior of strong (external or insider) attackers. Finally, we believe that our proposed methods can be applied for verifying other similar systems, which we will show in our follow-up work.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] J. Yick, B. Mukherjee, and D. Ghosal, "Wireless sensor network survey," *Computer Networks*, vol. 52, no. 12, pp. 2292–2330, 2008.

[2] B. Marchi, A. Grilo, and M. Nunes, "DTSN: distributed transport for sensor networks," in *Proceedings of the 12th IEEE International Symposium on Computers and Communications (ISCC '07)*, pp. 165–172, Aveiro, Portugal, July 2007.

[3] L. Buttyan and A. M. Grilo, "A secure distributed transport protocol for wireless sensor networks," in *Proceedings of the IEEE International Conference on Communications (ICC '11)*, pp. 1–6, Kyoto, Japan, June 2011.

[4] L. Buttyan and L. Csik, "Security analysis of reliable transport layer protocols for wireless sensor networks," in *Proceedings of the IEEE Workshop on Sensor Networks and Systems for Pervasive Computing (PerSeNS '10)*, pp. 1–6, Mannheim, Germany, March 2010.

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

[6] J. Goubault-Larrecq, C. Palamidessi, and A. Troina, "A probabilistic applied pi-calculus," in *Programming Languages and Systems*, pp. 175–190, Springer, 2007.

[7] P. R. D'Argenio and E. Brinksma, "A calculus for timed automata," Tech. Rep., Theoretical Computer Science, 1996.

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

[9] T. Dierks and E. Rescorla, "The transport layer security (TLS) protocol version 1. 2," RFC 5246, Internet Engineering Task Force, 2008.

[10] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, parts I and II," *Information and Computation*, vol. 100, no. 1, pp. 1–77, 1992.

[11] T. V. Thong and A. Dvir, "On formal and automatic security verification of wsn transport protocols," Tech. Rep. 2013/014, Cryptology Eprint Archive, 2013.

[12] R. Lanotte, A. Maggiolo-Schettini, and A. Troina, "Weak bisimulation for prob-abilistic timed automata," in *Proceedings of the International Conference on Software Engineering and Formal Methods (SEFM '03)*, pp. 34–43, IEEE CS Press, 2003.

[13] B. Blanchet, "Automatic proof of strong secrecy for security protocols," in *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pp. 86–100, Oakland, Calif, USA, May 2004.

[14] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and J. D. Tygar, "SPINS: security protocols for sensor networks," in *Proceedings of the 7th ACM Annual International Conference on Mobile Computing and Networking (MobiCom '01)*, pp. 189–199, Rome, Italy, July 2001.

[15] J. Bengtsson and F. Larsson, "Uppaal a tool for automatic verification of real-time systems," Tech. Rep., Uppsala University, 1996.

[16] J. D. Marshall II and X. Yuan, "An analysis of the secure routing protocol for mobile ad hoc network route discovery: Using intuitive reasoning and formal verification to identify flaws," Tech. Rep., The Florida State University, 2003.

[17] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4-0: verification of probabilistic realtime systems," in *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806 of *Lecture Notes in Computer Science*, pp. 585–591, Springer, 2011.

[18] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[19] G. Holzmann, *The Spin Model Checker: Primer and Reference Manual*, Addison-Wesley Professional, 1st edition, 2003.