Formal Security Verification of Transport Protocols for Wireless Sensor Networks

Vinh-Thong Ta[1], Amit Dvir[4], and Levente Buttyán[2,3]

INRIA, CITI/INSA-Lyon, F-69621, Villeurbanne, France[1]
Laboratory of Cryptography and System Security (CrySyS), BME, Hungary[2]
MTA-BME Information Systems Research Group,
Magyar tudósok körútja 2, 1117 Budapest, Hungary[3]
Computer Science School, The College of Management - Academic Studies, Israel[4]
vinh-thong.ta@inria.fr, azdvir@gmail.com, buttyan@crysys.hu

Abstract. In this paper, we address the problem of formal security verification of transport protocols for wireless sensor networks (WSN) that perform cryptographic operations. Analyzing this class of protocols is a difficult task because they typically consist of complex behavioral characteristics, such as launching timers, performing probabilistic behavior, and cryptographic operations. Some of the recently published WSN transport protocols are DTSN, which does not include cryptographic security mechanism, and two of its secured versions, SDTP and STWSN[1]. In our previous work, we formally analyzed the security of Distributed Transport for Sensor Networks (DTSN) and Distributed Transport Protocol for Wireless Sensor Networks (SDTP), and showed that they are vulnerable against packet modification attacks. In another work we proposed a new Secure Transport Protocol for WSNs (STWSN), with the goal of eliminating the vulnerability of DTSN and SDTP, however, its security properties have only been informally argued. In this paper, we apply formal method to analyze the security of STWSN.

1 Introduction

Wireless Sensor Networks [2] consist of a large number of resource constrained sensor nodes and a few more powerful base stations. The sensors collect various types of data from the environment and send those data to the base stations using multi-hop wireless communications. Some typical applications that require the use of a transport protocol for ensuring reliable delivery and congestion control are: reliable control and management of sensor networks; remotely programming/retasking sensor nodes over-the-air. It is widely accepted that transport protocols used in wired networks (e.g., the well-known TCP) are not applicable in WSNs, because they perform poorly in a wireless environment and they are

[1] STWSN and SDTP$^+$ are the same. SDTP$^+$ is the protocol name used in [1], however, because it is not only a minor extension of SDTP, but is based on a completely different security solutions, it is better to change the name to STWSN.

not optimized for energy consumption. Therefore, a number of transport protocols specifically designed for WSNs have been proposed in the literature (see, e.g., [3] for a survey). The main design goal of those transport protocols is to achieve reliability and energy efficiency. However, despite the fact that WSNs are often envisioned to operate in hostile environments, existing transport protocols for WSNs do not address security issues at all and, as a consequence, they ensure reliability and energy efficiency only in a benign environment where no intentional attack takes place [4].

Attacks against WSN transport protocols can be attacks against reliability and energy depleting attacks. In the first case, the attackers can cause undetected or permanent packet lost, while in the second case the attackers make the sensor nodes deplete their battery by performing futile computations. Some of the recently published WSN transport protocols are DTSN [5], and two of its security extensions, SDTP [6] and STWSN [1]. Many tricky attack scenarios have been found against WSN transport protocols, and even secure WSN transport protocols that were believed to be secure, have turned out to be vulnerable. The main reason is that the designers reason about the security of their protocol based only on manual analysis, which is error-prone. DTSN and SDTP have been analyzed based on formal method in [7], and were shown to be vulnerable against packet modification attacks.

STWSN is designed with the goal of eliminating the vulnerability of DTSN and SDTP, however, its security properties have been argued only informally. Our goal is to apply formal method to analyze WSN transport protocols, which provide a more reliable and systematic way of security proof. In this paper, we apply formal method to analyze the security of WSN transport protocol. In particular, using the process algebra language, $crypt^{prob}_{time}$, proposed in [7], we showed that the STWSN protocol is secure against packet modification attacks.

2 Related Works

We provide a brief overview of the DTSN and SDTP protocols, which are two recent and representative WSN transport protocols, and they are closely related to the STWSN protocol.

DTSN [5] 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 the end-to-end retransmission mechanism is that it allows intermediate nodes to cache and retransmit data packets, hence, the average number of hops a retransmitted data packet must travel is smaller than the length of the route between the source and the destination. Intermediate nodes do not store all packets but only store packets with some probability p, which makes it more efficient. DTSN uses positive acknowledgements (ACKs), and negative acknowledgements ($NACK$s) to control caching and retransmissions. 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.

Reasoning about the security of 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.

SDTP [6] is a security extension of DTSN aiming at patching the security holes in DTSN. SDTP protects data packets with MACs (Message Authentication Code) computed over the whole packet. Each MAC is computed using a per-packet key correspond to each packet. The intermediate and source nodes cache packets along with their MACs, and whenever the destination wants to acknowledge the first n packets it sends an ACK message with the key for the MAC corresponding to this packet. Similarly, when the destination request the retransmission of packets it sends a $NACK$ message with the keys corresponding the given packets. The rationale behind the security of SDTP is that only the source and destination knows the correct keys to be revealed.

Reasoning about the security of SDTP: The main security weakness of the SDTP protocol is that the intermediate nodes store the received data packets without any verification. Intermediate nodes do not verify the origin and the authenticity of the data packets or the ACK and the $NACK$ messages, namely, they cannot be sure whether the data packets that they stored were sent by the source node, and the control messages were really sent by the destination. Indeed, the security solution of SDTP only enables intermediate nodes to verify the matching or correspondence of the stored packets and the revealed $ACK/NACK$ keys. Hence, SDTP can be vulnerable in case of more than one attacker node (compromised node) who can cooperate.

In [7] we formally proved that an attacker may cause permanent loss of some data packets both in DTSN and STDP, by forging or altering the sequence number in ACK message. We also showed that an attacker can trigger futile retransmission of the corresponding data packets by either setting bits in the bit map of the $NACK$ packets or forging/altering $NACK$ packets. The first case violates the reliability while the second case violates the energy efficiency requirement. In [1] we proposed a new secured WSN transport protocol, called STWSN, and argued its security. However, the security of STWSN has not been analyzed based on formal method so far. In this paper, we perform formal secu-

rity analysis of STWSN and prove that it is secure against the successful attack scenarios in case of DTSN and SDTP.

3 STWSN - A Secure Distributed Transport Protocol for WSNs

We proposed STWSN in [1], in order to patch the security weaknesses can be found in DTSN and SDTP. STWSN aims at authenticating and protecting the integrity of control packets, and is based on an efficient application of digital signature and authentication values, which are new compared to SDTP. The security mechanism of STWSN is based on the application of Merkle-tree [8] and hash chain [9], which have been used for designing different security protocols such as Castor [10], a scalable secure routing protocols for ad-hoc networks, and Ariadne [11]. Our contribution is applying Merkle-tree and hash chain in a new context. The general idea of STWSN is the following: two types of "per-packet" *authentication values* are used, ACK and $NACK$ authentication values. The ACK authentication value is used to verify the ACK packet by any intermediate node and the source, whilst the $NACK$ authentication value is used to verify the $NACK$ packet by any intermediate node and the source. The ACK authentication value is an element of a hash chain [9], whilst the $NACK$ authentication value is a leaf and its corresponding sibling nodes along the path from the leaf to the root in a Merkle-tree [8]. Each data packet is extended with one Message Authentication Code (MAC) value (the MAC function is HMAC), instead of two MACs as in SDTP. STWSN adopt the notion and notations of the pre-shared secret S, ACK, and $NACK$ master secrets K_{ACK}, K_{NACK}, which are defined and computed in exactly the same way as in SDTP [6]. However, in STWSN the generation and management of the per-packet keys $K^{(n)}_{ACK}$, $K^{(n)}_{NACK}$ is based on the application of hash-chain and Merkle-trees, which is different from SDTP.

3.1 The *ACK* Authentication Values

The ACK authentication values are defined to verify the authenticity and the origin of ACK messages. The number of data packets that the source wants to send in a given session, denoted by m, is assumed to be available. At the beginning of each session, the source generates the ACK master secret K_{ACK} and calculates a hash chain of size $(m+1)$ by hashing K_{ACK} $(m+1)$ times, which is illustrated in Figure 1. Each element of the calculated hash-chain represents a *per packet ACK authentication value* as follows: $K^{(m)}_{ACK}, K^{(m-1)}_{ACK} ..., K^{(1)}_{ACK}, K^{(0)}_{ACK}$, where $K^{(i)}_{ACK} = h(K^{(i+1)}_{ACK})$ and h is a one-way hash function. The value $K^{(0)}_{ACK}$ is the root of the hash-chain, and $K^{(i)}_{ACK}$ represents the ACK authentication value corresponding to the packet with sequence number i. When the destination wants to acknowledge the successful delivery of the i-th data packet, it reveals the corresponding $K^{(i)}_{ACK}$ in the ACK packet.

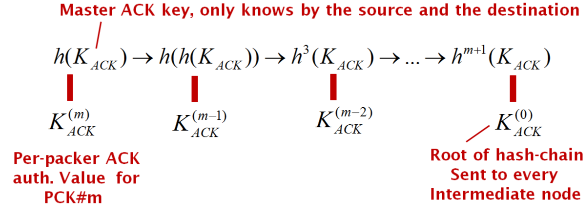

Fig. 1. *The element $K^{(i)}_{ACK}$, $i \in \{1, \dots, m\}$, of the hash-chain is used for authenticating the packet with the sequence number i. The root of the hash-chain, $K^{(0)}_{ACK}$, which we get after hashing $(m+1)$ times the ACK master key K_{ACK}. This root is sent to every intermediate node in the open session packet, which is digitally signed by the source.*

3.2 The *NACK* Authentication Values

For authenticating the *NACK* packets, STWSN applies a Merkle-tree (also known as hash-tree), which is illustrated in Figure 2. When a session has started, the source computes, the *NACK* per-packet keys $K^{(n)}_{NACK}$ for each packet to be sent in a given session. Afterwards, these *NACK* per-packet keys are hashed and assigned to the leaves of the Merkle-tree: $K^{'(n)}_{NACK} = h(K^{(n)}_{NACK})$. The internal nodes of the Merkle-tree are computed as the hash of the (ordered) concatenation of its children. The root of the Merkle-tree, $H(h_p, S_p)$, is sent by the source to intermediate nodes in the same open session packet that includes the root of the hash-chain. For each $K^{(j)}_{NACK}$, $j \in \{j1, \dots, j_m\}$, the so called sibling values $S^j_1, \dots, S^j_t$, for some t, are defined such that the root of the Merkle-tree can be computed from them. For instance, the sibling values of $K^{(j1)}_{NACK}$ are $K^{(j2)}_{NACK}$, $S_1, \dots, S_p$. From these values $H(h_p, S_p)$ can be computed.

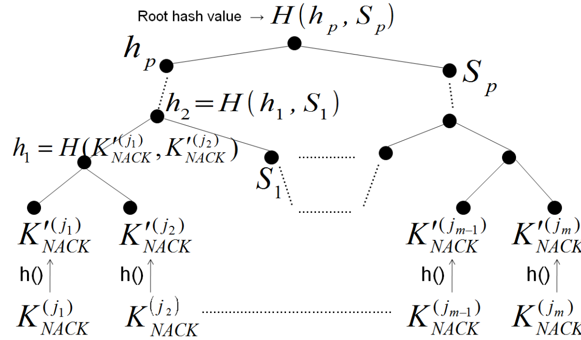

Fig. 2. *The structure of Merkle-tree used in STWSN. Each internal node is computed as the hash of the ordered concatenation of its children.* The root of the tree, $H(h_p, S_p)$, is sent out by the source.

3.3 The operation of STWSN

In this section a short flow of the STWSN protocol is given, for more information please refer to [1]. When a session is opened, first, the source computes the ACK and $NACK$ master keys K_{ACK} and K_{NACK}, respectively. Then, the source calculates the hash-chain and the Merkle-tree for the session. Afterward, the source sends an open session message with the following parameters: the roots of the hash chain ($K_{ACK}^{(0)}$) and of the Merkle-tree ($H(h_p, S_p)$), the length of the hash chain ($m+1$), the session *SessionID*, the source and destination IDs. Before sending the open session packet, the source digitally signs it to prevent the attackers from sending fake open session packets. When the destination node receives an open-session packet sent by the source, it verifies the signature computed on the packet. Upon success, the destination starts to generate the same values as the source and sends an ACK packet to the source. Upon receipt of an open session packet and the corresponding ACK packet, an intermediate node verifies signature computed on the packet, and in case of success, it stores the root values of the hash chain and the Merkle-tree, the session ID, *SessionID*, and forwards the packet towards the destination. Otherwise, an intermediate node changes its probability to store packets in the current session to zero.

After receiving an ACK message corresponding to the session open packet, from the destination, the source starts to send data packets. Each data packet is extended with the MAC, computed over the whole packet (except for the *EAR* and *RTX* flags), using the shared secret between the source and the destination. Upon receipt of a data packet, an intermediate node stores with probability p the data packet and forwards the data packet towards the destination. Upon receiving a data packet with sequence number i, first, the destination checks MAC using the secret shared between the source and the destination. Upon success, the destination delivers the packet to the upper layer. Otherwise, the packet is ignored and dropped. Upon the receipt of a packet with a set *EAR* flag, the destination sends an ACK or a $NACK$ packet depending on the existence of gaps in the received data packet stream. The ACK packet that refers to sequence number i is composed of the pair (i, $K_{ACK}^{(i)}$). Similarly, the $NACK$ packet with base sequence number i is extended with the ACK authentication value ($K_{ACK}^{(i)}$), and if the destination wants to request for re-transmission of some packet j, then it also includes the corresponding $NACK$ authentication values $K_{NACK}^{(j)}$, $S_1^j, \dots,$ S_q^j in the $NACK$ packet.

When an intermediate node receives an ACK packet, (i, $K_{ACK}^{(i)}$), it verifies the authenticity and the origin of the ACK message by hashing $K_{ACK}^{(i)}$ i times, and comparing the result with the stored root value of the hash chain. If the two values are equal, then all the stored packets with the sequence number less than or equal to i are deleted. Afterward, the intermediate node passes on the ACK packet towards the source. Otherwise, the ACK packet is ignored and dropped. In case of $NACK$ packet that refers to the sequence numbers i, the same is perform for the ACK part. From the $NACK$ part the root of the Merkle-tree is re-generated, and in case of equality, the stored data packets are re-transmitted

and the $NACK$ is modify. Afterward, forwards the $NACK$ with the modified list towards the source. When the source node receives an ACK packet or $NACK$ packets it perform the same steps as the intermediate (with the fact the $p = 1$).

In DTSN and SDTP, the destination sends an ACK or a $NACK$ packet upon receipt of an EAR. In order to mitigate the effect of EAR replay or EAR forging attacks where the EAR flag is set/unset by an attacker(s), STWSN uses two new mechanisms: status timer and limiting the number of responses to EARs. The status timer is set at the destination and its duration could be a function of the source EAR timer. To counter that attackers always set the EAR bits, the destination limits the number of responses on receiving a set EAR flag. In the period of the destination's EAR timer the destination will not send more than X control packets.

4 Formal security analysis of STWSN using $crypt_{time}^{prob}$

In this subsection, we perform a formal security analysis of STWSN based on a mathematically sound formal language, the $crypt_{time}^{prob}$ calculus [7], which has been used to analyze the security of DTSN and SDTP. We start with a brief description of $crypt_{time}^{prob}$, and provide the proof technique based on it, finally, we turn to prove the security of our proposed STWSN protocol.

4.1 The $crypt_{time}^{prob}$ calculus

Due to lack of space we only provide a very brief overview of the $crypt_{time}^{prob}$ calculus, interested readers are referred to [7] for further details. $crypt_{time}^{prob}$ is a probabilistic timed calculus, for modeling and analyzing cryptographic protocols that involve clock timers and probabilistic behavioral characteristics. $crypt_{time}^{prob}$ has been successfully used for analyzing the security of DTSN and SDTP [7].

The basic concept of $crypt_{time}^{prob}$ is inspired by the previous works [12], [13], [14] proposing analysis methods separately for cryptographic, timed, and probabilistic protocols, respectively. Specifically, it is based on the concept of probabilistic timed automata, hence, the correctness of $crypt_{time}^{prob}$ comes from the correctness of the automata because the semantics of $crypt_{time}^{prob}$ is equivalent to the semantics of the probabilistic timed automata.

The formal syntax of $crypt_{time}^{prob}$ is composed of two main building blocks, namely, the *terms* which model protocol messages and their components, and *probabilistic timed processes* which describe the internal operation of communication partners according to their specification. Terms can be, for example, random nonces, secret keys, encryptions, hashes, MACs and digital signatures computed over certain messages, and they can represent entire messages as well. The set of terms also includes communication channels (denoted by c_i, for some index i) defined between participants, such that messages can be sent and received through these channels. We distinguish *public* and *private* channels, where the attackers can eavesdrop on public channels, while they cannot in the private case. The set of probabilistic timed processes defines the internal behavior of

the participants, namely, each process defines an action can be performed by a given participant. For instance, *input* and *output processes* define the message receiving and sending actions, respectively. Processes also define the message verification steps that a communication partner should perform on the received messages (e.g., signature verification, comparisons, MAC verification). Finally, processes can also be used to define the whole protocol, composing of several communication partners running parallel. This is similar to the terminology of sub-procedures and main-procedure in programming languages (e.g., C, Java). We denote probabilistic timed processes by $procA^i$ with $i \in \{1, 2, \ldots, k\}$ for some finite k, and A can be the name of a communication partner or a protocol (e.g., *procSrc* represents the process that describes the behavior of the source node). Non-channel terms are given the name of a message or its components (e.g., ID is a term that models a message ID).

The operational semantics of $crypt^{prob}_{time}$ is built-up from a probabilistic timed labeled transition system (PTTS) defined specifically for this calculus. The PTTS contains the rules of form $s_1 \stackrel{\alpha,\, d}{\rightarrow} s_2$, where s_1 describes the current state of a given process, while s_2 represents the state we reach after some action α has been performed, which consumes d time units. For instance, α can be the message sending action, while s_1 and s_2 are the states before and after the message has been sent, respectively. There are three types of actions that can be performed by the communication partners: (i) silent (internal computation) action; (ii) message input or (iii) message output on a channel. Silent action (e.g., message verification steps) are not visible for an external observer (environment), while message input and outputs on a public channel are visible. Message input and output on a private channel can be seen as silent actions. The PTTS of $crypt^{prob}_{time}$ contains the rules that define all the possible actions of the communication partners, according to the protocol description.

In order to prove or refute the security of protocols and systems, $crypt^{prob}_{time}$ is equipped with the *weak probabilistic timed (weak prob-timed) labeled bisimilarity.* The bisimilarity definition is used to prove or refute the behavioral equivalence between two variants of a protocol or system. The definition of weak prob-timed bisimilarity is given as follows:

Definition 1 (***Weak prob-timed labeled bisimulation for*** $\boldsymbol{crypt^{prob}_{time}}$)
We say that two states $s_1 = (procA^1,\, v_1)$ *and* $s_2 = (procA^2,\, v_2)$ *are weak prob-timed labeled bisimilar, denoted by* $(s_1\ \Re^p_t\ s_2)$

1. *if an observer who can eavesdrop on the network communication cannot distinguish the message output or input in states* s_1 *and* s_2 *(which we called as statically equivalence [7]);*
2. *if from* s_1 *we can reach the state* s'_1 *after a silent (internal) action after* d_1 *time units, then* s_2 *can simulate this action via the corresponding silent action trace after* d_2 *time units, leading to some* s'_2*, and* $s'_1\ \Re^p_t\ s'_2$ *holds again.*
3. *if from* s_1 *we can reach the state* s'_1 *after a non-silent labeled transition (i.e., message input or ouput) after* d_1 *time units, then* s_2 *can simulate this action via the corresponding labeled transition trace after* d_2 *time units, leading to some* s'_2*, and* $s'_1\ \Re^p_t\ s'_2$ *holds again,*

and vice versa. We say that two variants of a protocol Prot1 and Prot2 are weak prob-timed labeled bisimilar if their initial states (s_1^{init}, s_2^{init}) are weak prob-timed labeled bisimilar.

In this definition, s_j is a protocol (system) state, which is composed of the pair ($procA^j$, v_j), where $procA^j$ is a $crypt_{time}^{prob}$ process, representing the current behavior of the participants in the protocol, and v_j is the current timing value(s) of the clock(s). Hence, based on this definition whenever we refer to a (prob. timed) protocol A we mean the state s^{init}, $s^{init} = (procA^{init}, v^{init})$.

Intuitively, Definition 1 says that two versions of a given protocol are "behavioral" equivalent if any action (formally, any labeled transition) that can be performed by one protocol version can be simulated by the corresponding action(s) (formally, a corresponding trace of labeled transition) in the another version, and vice versa. More precisely, if s_1 and s_2 are in weak prob-timed labeled bisimulation, then the behavior of the two protocol versions $procA^1$ and $procA^2$ are equivalent for an observer (environment) who eavesdrops on the communication between every pair of partners.

4.2 Security proof technique based on $crypt_{time}^{prob}$

We apply the proof technique that is based on Definition 1. Namely, we define an ideal version of the protocol run, in which we specify the ideal (i.e., 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, regardless of the attackers' activity. Then, we examine whether the real and the ideal versions, running in parallel with the same attacker(s), are weak prob-timed bisimilar.

Definition 2 *Let the $crypt_{time}^{prob}$ processes procProt and $procProt^{ideal}$ specify the real and ideal versions of some protocol Prot, respectively. We say that Prot is secure (up to the ideal specification) if ($procProt$, v^{init}) and ($procProt^{ideal}$, v_{ideal}^{init}) are weak prob-timed bisimilar:*

$$(procProt,\ v^{init}) \approx_{pt} (procProt^{ideal},\ v_{ideal}^{init}),$$

where v^{init} and v_{ideal}^{init} are the initial values of the clocks (typically in reset status). The strictness of the security requirement, which we expect a protocol to fulfill, depends on how "ideally secure" we specify the ideal version. Intuitively, Definition 2 says that *Prot* is secure if the attackers cannot distinguish the operation of the two instances based on the message outputs (and inputs). In the rest of this section, we refer to the source, intermediate and destination nodes as S, I and D, respectively.

Let us consider a simplified network topology for the STWSN protocol. We assume the network topology $S{-}I{-}D$, where " $-$ " represents a bi-directional link. Moreover, public communication channels are defined between each node pair for message exchanges, c_{si} between S and I, c_{id} between I and D. We also

assume the presence of an attacker or attackers who can eavesdrop on public communication channels, and can use the eavesdropped information in their attacks (e.g., modifying the ACK/NACK packets and forward them).

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 private (hidden) channels between honest parties, on which the communication cannot be observed by the attacker(s). Figure 3 shows the difference in more details. In the ideal case, three private channels are defined which are not available to the attacker(s). 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, hence, when I receives a packet via c_{si} it compares the message with *pck*. The same happens when I sends a packet to D, and vice versa, from D to I and I to S. Whenever a honest node receives an unexpected data (i.e., not the same as the data received on private channel), 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.

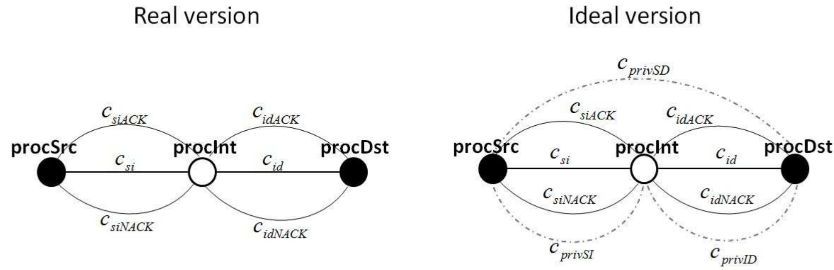

Fig. 3. *procSrc*, *procInt* and *procDst* denote the $crypt^{prob}_{time}$ processes that specify the operation of the source, the intermediate and the destination nodes, respectively. In the ideal version, the private (hidden) channels c_{privSD}, c_{privID} and c_{privSI} are defined between *procSrc* and *procDst*, *procInt* and *procDst*, *procSrc* and *procInt*, respectively.

With this definition of the ideal system we ensure that the source and intermediate nodes are not susceptible to the modification or forging of data packets and $ACK/NACK$ 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 $\mathcal{M}_A$: We assume that the attacker(s) can intercept the information output by the honest nodes on public channels, and modify them according to its (their) knowledge and computation ability. The attacker's knowledge consists of the intercepted output messages during the protocol run and the information it can create. The attacker(s) can modify the elements of the $ACK/NACK$ messages, such as the acknowledgement number and the bits

for retransmissions, as well as the EAR and RTX bits, and sequence numbers in data packets. The attacker can also create entire data or control packets based on the data it possesses. Further, an attacker can send packets to its neighborhood. We also consider several attackers who can share information with each other.

To describe the activity of the attacker(s), we apply the concept based on the so-called *environment*, used in the applied π-calculus [12] to model the presence of the attacker(s). 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. This definition of attackers allow us to apply the weak prob-timed bisimilarity in our security proofs.

4.3 Security analysis of STWSN based on the $crypt^{prob}_{time}$ calculus

As already mentioned in Section 3, STWSN uses different cryptographic primitives and operations such as one-way hash, message authentication code (MAC), and digital signature. In $crypt^{prob}_{time}$, secret and public keys can be defined by (atomic) names, and cryptographic operations can be defined as functions (for modelling crypto-primitive generation) and equations (for cryptographic verification):

Keys: sk_{src}, pk_{src}, K_{ack}; K_{nack}; K_{sd};

Functions: $sign(t,\, sk_{src})$; $H(t)$;
Equation: $checksign(sign(t,\, sk_{src}),\, pk_{src}) = ok$;

Functions: $K(n,\, ACK)$; $K(n,\, NACK)$;
$mac(t,\, K(n,\, ACK))$; $mac(t,\, K(n,\, NACK))$;
Equations:
$checkmac(mac(t,\, K(n,\, ACK)),\, K(n,\, ACK)) = ok$;
$checkmac(mac(t,\, K(n,\, NACK)),\, K(n,\, NACK)) = ok$.

where sk_{src} and pk_{src} represent the secret and public key of the source node. K_{ack}, K_{nack} and K_{sd} represent the $ACK/NACK$ master keys, and the shared key of the source and the destination for a given session, which are freshly generated at the beginning of each session. The functions $sign(t,\, sk_{src})$ and $H(t)$ define the digital signature computed on the message t using the secret key sk_{src}, and the one-way hash computed on t, respectively. The equation $checksign(sign(t,\, sk_{src}),\, pk_{src}) = ok$ defines the signature verification, using the corresponding public key pk_{src}. We do not define an equation for the hash function $H(t)$ in order to ensure its one-way property. Namely, $H(t)$ does not have a corresponding inverse function which returns t, and $H(t_1) = H(t_2)$ holds only when t_1 and t_2 are the same.

For simplicity, we assume the network topology $S{-}I{-}D$, where " $-$ " represents a bi-directional link, while S, I, D denote the source, an intermediate

node, and the destination node, respectively (like in Figure 3). However, we emphasize that the security proofs in this simplified topology can also be applied (with some extension) and remain valid in the topologies that contain more intermediate nodes. We define symmetric public channels between the upper layer and the source, c_{sup}; the upper layer (i.e. the application that uses the protocol) and the destination, c_{dup}; the source and the intermediate node, c_{si}; the intermediate node and the destination c_{id}. The public channels c_{siACK}, c_{siNACK}, c_{idACK} and c_{idNACK} are defined for sending and receiving ACK and $NACK$ messages between the source and the intermediate, and between the destination and the intermediate nodes, respectively. Public channels c_{siOPEN} and c_{idOPEN} are defined for delivering the open-session messages between S and I, as well as I and D. To apply Definition 1 in our proofs, additional public channels $c_{badOPEN}$, c_{badPCK}, c_{error} and $c_{sessionEND}$ are defined for sending and receiving *bad packet*, *error* and *session-end* signals. Honest nodes send out these signals when error or session-end is detected at these nodes. Finally, we add a public channel c_{emptyC} for signalling that the cache has been emptied at a given node.

According to the proof technique, we define an ideal and a real version of the STWSN protocol. In the ideal version of the STWSN protocol the definition of the processes *procSrc*, *procInt*, and *procDst* in $crypt^{prob}_{time}$ are extended (compared to the real version) with some additional equality checks between the messages received on the corresponding private and public channels. Specifically, processes *procDst* and *procInt* output the special constant *BadOpen* on channel $c_{badOPEN}$ when they receive an unexpected open-session packet (i.e., when the message which the honest nodes receive on the public channels c_{siOPEN} and c_{idOPEN} is not equal to the corresponding packet received on the private channels c_{privSI} and c_{privID}, respectively). Process *procDst* outputs the constant *BadData* on c_{badPCK} when it receives an unexpected data packet. After receiving an unexpected $ACK/NACK$, the processes *procSrc* and *procInt* output the constant *BadControl* on c_{badPCK}. Finally, after emptying the buffer *procSrc* and *procInt* output the constants *EmptyCacheS* and *EmptyCacheI*, respectively.

We examined the security of STWSN regarding the following scenarios: In the first scenario SC-1, the attacker(s) modifies the ACK/NACK messages, while in the SC-2 the attacker(s) modifies the EAR and RTX bits in data packets. Let us name the $crypt^{prob}_{time}$ processes that define the real and ideal versions of STWSN by $procSTWSN$ and $procSTWSN^{ideal}$, respectively.

– **Scenario SC-1**: STWSN is not vulnerable to the attack scenario SC-1 because process $procSTWSN^{ideal}$ can simulate (according to Definition 1) every labeled transition produced by $procSTWSN$: Recall that in STWSN S verifies the $ACK/NACK$ packets by comparing the stored roots of the hash-chain and the Merkle-tree with the re-computed roots. For simplicity, we assume that the source is storing the first three packets for a given session, and the source wants to send four data packets in this session. This means that the root of the hash-chain is $H(H(H(H(K_{ack}))))$. Let assume that the source has received an ACK packet, where $ACK = (m, h_m)$, and according to the protocol, to make the source accept this ACK the m-time hashing

on h_m must be $H(H(H(H(H(K_{ack})))))$. To empty the buffer, m must be at least 3. In case $m = 4$, h_m must be $H(K_{ack})$. This hash value cannot be computed by the attacker(s) because the source and the destination never reveal K_{ack}, and function $H(t)$ is defined to be one-way, so from $H(H(K_{ack}))$ the attacker cannot compute $H(K_{ack})$. Hence, the attacker(s) must receive or intercept $H(K_{ack})$ from a honest node, which means that $H(K_{ack})$ has already been revealed by the destination. m cannot be greater than four, otherwise, the attacker(s) must have K_{ack}, that is, K_{ack} must have been revealed by the source or destination, which will never happen according to the protocol. When $m = 3$, h_m must be $H(H(K_{ack}))$, which cannot be computed by the attacker(s) since according to the protocol, K_{ack} and $H(K_{ack})$ have not been revealed yet, and we did not define any equation for the hash function $H(t)$, hence, from $H(h_i)$ the value of h_i cannot be derived. To summarize, either the attacker sends a correct ACK or the ACK with incorrect authentication value, the ideal and the real systems can simulate each other. In the first case, the constant *EmptyCacheS*, while in the second case *BadControl* is output in both systems. The reasoning is similar in case of $NACK$ message.

In the following, we examine whether the attackers can make the intermediate node incorrectly empty its buffer. Let us assume that I has already accepted the open session packet and has stored the hash-chain root, denoted by h_{root}, in it. The signature in the open session packet must be computed with the secret key of the source, sk_{src}. This is because after receiving a packet on channel c_{siOPEN}, the verification $[checksign(x_{sig},\ pk_{src}) = ok]$ is performed within process *procInt*, and only the signature computed with sk_{src} can be verified with pk_{src}. However, this means that h_{root} must be generated by the source, that is, $h_{root} = H(H(H(H(H(K_{ack})))))$. Again, assume that in the current state I stores the first three data packets. From this point, the reasoning is similar to the source's case, namely, either the constant *EmptyCacheI* or *BadControl* is output in both the ideal and the real systems.

– **Scenario SC-2**: The formal proof regarding the second scenario is based on a similar concept as in case of analyzing fault tolerance systems. Namely, for reasoning about the scenario *SC-2*, we modify the ideal version of STWSN as follows: we limit the number of the $ACK/NACK$ that the destination sends when it receives a data packet in which the *EAR* flag is set to 1 by the attacker. Based on the concept of private channels where the destination is informed about the message sent by a honest node, in the ideal system, the destination is able to distinguish between the *EAR* flag set by the attacker and the flag set by a honest node. In the ideal system, we modify the specification of the destination such that within a session, the destination will only handle according to the protocol, the first MAX_{badear} packets in which the *EAR* flag is set to 1 by the attacker, and sends back the corresponding $ACK/NACK$ for them. Formally, for the first MAX_{badear} packets with an incorrectly set *EAR* flag, the destination does not output the constant *Bad-*

Data on channel c_{badPCK}, but only from the following incorrect packet. The constant MAX_{badear} is an application specific security treshold.
In the STWSN protocol, to alleviate the impact of the *EAR* setting attack, the destination limits the number of responses for the packets with the *EAR* flag set by either an attacker or a honest node [1]. Within a finite period of time, called *destination EAR timer* (denoted by *dest_EAR_timer*), the destination node will not send more than D control packets in total, for some given security treshold D. Within a session, the destination launches the timer *dest_EAR_timer* when it has received the first packet containing a set *EAR* flag. Until the session ends, *dest_EAR_timer* is continually reset upon timeout. Let tmr_{dst} be the upperbound of the number of launching/resetting *dest_EAR_timer* within a session. The values of *dest_EAR_timer* and D are set such that $D{\times}tmr_{dst} \leq MAX_{badear}$.

The main difference between the ideal and the real STWSN specifications, *procSTWSN* and $procSTWSN^{ideal}$, is that in the ideal case the destination does not need to launch the timer *dest_EAR_timer*, because it is aware of the packets with an incorrectly set *EAR* flag. Instead, the destination only limits the total number of responses for the incorrect packets to MAX_{badear}. To prove the security of STWSN regarding *SC-2*, we prove that the STWSN protocol is secure regarding the scenario *SC-2*, by showing that $procSTWSN^{ideal}$ weak prob-timed simulates the real system *procSTWSN* (instead of resisting on bisimilarity). Intuitively, this means that the set of probabilistic timed transitions of *procSTWSN* is a subset of the set of probabilistic timed transitions of $procSTWSN^{ideal}$. The reverse direction that *procSTWSN* can simulate $procSTWSN^{ideal}$, is not required for this scenario.

We also examined the other possible attack attempts (e.g., when the attacker(s) modifies the data part in data packets, and the open-session packets) with the same method, based on the definition of weak prob-timed bisimilarity. Namely, we define the corresponding real and ideal versions, and we showed that they can simulate each other regarding the outputs of the constants *BadData* and *BadOpen*, respectively. Hence, based on the Definition 1, we showed that the two versions are weak prob-timed bisimilar. Due to lack of space we do not include them here, interested readers can find them in the report [15].

5 Conclusion and Future works

We addressed the problem of formal security verification of WSN transport protocols that launch timers, as well as performing probabilistic and cryptographic operations. We argued that formal analysis of WSN transport protocols is important because informal reasoning is error-prone, and due to the complexity of the protocols, vulnerabilities can be overlooked. An example would be the case of SDTP, which was believed to be secure, but later, it turned out to be vulnerable. In this paper, using the algebra language $crypt^{prob}_{time}$, we formally proved that the STWSN protocol is secure against the packet modification attacks.

In this paper, we also demonstrated the expressive power of the $crypt^{prob}_{time}$ calculus, and showed that it is well-suited for analyzing protocols that may include timers, probabilistic behavior, and cryptographic operations. One interesting future direction could be examining the usability of $crypt^{prob}_{time}$ for other class of protocols (e.g., wired transport protocols). In addition, designing an automated verification method based on $crypt^{prob}_{time}$ also raises interesting questions.

References

1. A. Dvir, L. Buttyán, and V.-T. Ta. SDTP+: Securing a distributed transport protocol for wsns using merkle trees and hash chains. In *IEEE International Confenrence on Communications (ICC)*, pages 1–6, Budapest, Hungary, June 2013.
2. J. Yicka, B. Mukherjeea, and D. Ghosal. Wireless sensor network survey. *Computer Networks*, 52(12):2292–2330, Aug. 2008.
3. C. Wang, K. Sohraby, B. Li, M. Daneshmand, and Y. Hu. A survey of transport protocols for wireless sensor networks. *Network*, 20(3):34–40, May 2006.
4. L. Buttyán and L. Csik. Security analysis of reliable transport layer protocols for wireless sensor networks. In *IEEE Workshop on Sensor Networks and Systems for Pervasive Computing*, pages 1–6, Mannheim, Germany, March 2010.
5. B. Marchi, A. Grilo, and M. Nunes. DTSN - distributed transport for sensor networks. In *IEEE Symposium on Computers and Communications*, pages 165–172, Aveiro, Portugal, July 2007.
6. L. Buttyán and A. M. Grilo. A Secure Distributed Transport Protocol for Wireless Sensor Networks. In *IEEE International Conference on Communications*, pages 1–6, Kyoto, Japan, June 2011.
7. V.-T. Ta and A. Dvir. On formal and automatic security verification of wsn transport protocols. *ISRN Sensor Networks, accepted*, December 2013.
8. R. C. Merkle. Protocols for Public Key Cryptosystems. In *Symposium on Security and Privacy*, pages 122–134, California, USA, April 1980.
9. D. Coppersmith and M. Jakobsson. Almost optimal hash sequence traversal. In *Fourth Conference on Financial Cryptography*, pages 102–119, Southampton, Bermuda, March 2002.
10. W. Galuba, P. Papadimitratos, M. Poturalski, K. Aberer, Z. Despotovic, and W. Kellerer. Castor: Scalable Secure Routing for Ad-Hoc Networks. In *Infocom*, pages 1–9, Rio de Janeiro, Brazil, 2010.
11. Y.-C. Hu, A. Perrig, and D. B. Johnson. Ariadne: a secure on-demand routing protocol for ad hoc networks. *Wireless Networks Journal*, 11(1-2):21–38, 2005.
12. C. Fournet and M. Abadi. Mobile values, new names, and secure communication. In *ACM Symposium on Principles of Programming*, pages 104–115, 2001.
13. J. Goubault-larrecq, C. Palamidessi, and A. Troina. A probabilistic applied pi-calculus. In *Programming Languages and Systems*, volume 4807, pages 175–190. 2007.
14. P. R. D'Argenio and E. Brinksma. A calculus for timed automata. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135, pages 110–129. 1996.
15. V.-T. Ta and A. Dvir. On formal and automatic security verification of wsn transport protocols. Cryptology ePrint Archive, Report 2013/014, 2013.