Received ????; | Revised ????; | Accepted ?????

DOI: xxx/xxxx

ARTICLE TYPE

STWSN: A Novel Secure Distributed Transport Protocol for Wireless Sensor Networks

Amit Dvir*[1] | Vinh-Thong Ta[2] | Joseph (Sefi) Erlich[1] | Levente Buttyan[3,4]

[1]Ariel Cyber Innovation Center, Department of Computer Science, Ariel University, Israel
[2]School of Physical Sciences and Computing, University of Central Lancashire , Preston, UK
[3]Laboratory of Cryptography and System Security, BME, Budapest, Hungary
[4]MTA-BME Information Systems Research Group, BME, Budapest, Hungary

Correspondence
*Amit Dvir, Ariel University. Email: amitdv@ariel.ac.il

Summary

Several transport protocols for Wireless Sensor Networks (WSNs) have been designed to fulfill efficiency requirements such as energy and reliability. Unfortunately, most of these transport protocols do not include sufficient security mechanisms and hence, are vulnerable to numerous reliability and energy attacks. To address these vulnerabilities, this paper propose a novel secure transport protocol, named as Secure Transport Protocol for Wireless Sensor Networks (STWSN). Based on Distributed Transport for Sensor Networks (DTSN) protocol, our protocol add a new security extension in order to provide secure transport protocol. We provide both informal and formal security analyses of STWSN, and show that it resists attacks on energy efficiency and reliability requirements. Last but not least, a performance analysis based on the building blocks of STWSN, as well as some simulation results are also presented.

KEYWORDS:
Wireless sensor networks, Transport protocols, Hash-chain, Merkle-tree

1 | INTRODUCTION

Wireless Sensor Networks (WSN)s rely on low-cost devices with strong limitations such as energy and communications. One of the aims of WSN is to collect measurements over a given space and to transfer it to an external network via special nodes designated sink nodes. Due to the limitations of the devices, power saving techniques and low power communications for multihop data transmission are commonly implemented.

Some WSN applications demand a reliable transport layer protocol to ensure high end-to-end reliability. Some may require packet-driven reliability where the destination node have to receive all the packets which sent by the source node, while in order applications may require event-driven reliability, where the event must be detected.

Transport protocols for WSN require reliable delivery and congestion control. Unfortunately, although may transport protocols exist, they are focus on reliability and not on security. Hence, WSN transport protocols can be exposed to attacks which can be classified as attacks on reliability and energy depleting attacks. When attacker aim to drop a data packet in a way that the dropping remains undetected is reliability attack. In energy depleting attacks, the attacker triggers energy-intensive operations to deplete the nodes' batteries [1].

The STDP protocol [2], which is based on the DTSN protocol [3, 4], was the first transport protocol that focus on security for WSNs; however, in this paper we show that the STDP is still vulnerable because the security methods are not sufficient (see Section 3.2). To address these vulnerabilities, we propose a novel Secure Transport Protocol for Wireless Sensor Networks (STWSN), a secure extension for DTSN [4] (DTSN only provides reliability and energy efficiency in a benign environment).

1.1 | Our Contribution

To secure the WSN transport protocol, we apply hash-chains [5] and Merkle-trees [6] in a new context. Based on formal proofs, we show that the STWSN protocol is secure against attacks on DTSN and SDTP protocols. The STWSN protocol also integrates the following mechanisms to mitigate other attacks (a detailed explanation on these attacks can be found in Section 3):

- Aggregate timer - To mitigate and prevent energy attacks
- Status timer - To mitigate EAR replay or forging attacks
- Sending pre-deleted packets - To mitigate NACK bitmaps attacks
- Retransmission timers - To mitigate the impact of replay attacks
- Forwarding ACK packets after duplications - To mitigate NACK bitmaps attacks
- Limiting the retransmission number - To mitigate NACK replay attacks
- Limiting EAR responding - To mitigate EAR replay or forging attacks

We present a formal security analyses to demonstrate that STWSN resists attacks on energy efficiency and reliability requirements. Finally, an overhead analysis of the Merkle tree and hash chain is presented with performance analysis of our new protocol when compared to DTSN and STDP. In both our new protocol is much more secure and the overhead is not dramatic. A preliminary version of this work can be found in [7, 8]; however, here we added a substantial number of new contributions, such as the extension of the formal analysis, as well as more detailed protocol description details, overhead analysis and performance analysis sections.

The rest of the paper is organized as follows: Section 2 provides an overview of the most recent related works in this field. In Section 3 we provide a description of an attacker model, and discuss the main security vulnerabilities in the SDTP protocol. Our proposed STWSN protocol is detailed in Section 4, and an informal security analysis and formal security verification are found in Section 5, respectively. The STWSN performance analysis appears in Section 6 and concluding remarks and some possible future directions are presented in Section 7. Finally, the Appendixes presented in Sections 8 - 10.

2 | RELATED WORKS

The DTSN [3, 4] protocol is an efficient transport protocol designed for WSNs in which intermediate nodes cache data packets with some probability p, so that they can retransmit packets to the destination if needed. Hence, DTSN is more effective than transport protocols that apply end-to-end retransmission. Unfortunately, DTSN does not provide any protection for data and control packets, which makes it vulnerable to data and control packet modification/forging attacks. An attacker can arbitrarily change the content of the messages, causing permanent packet loss or session closing. More details on DTSN can be found in Section 8.

Ye et al. [9] proposed a mechanism that using automatic repeat request in order to improve reliability. The authors reported that the new mechanism exhibits good performance in several parameters such as reliable data transmission and end-to-end delay. Kordlar et al. [10] proposed a flexible recovery mechanism for a multipath forwarding mechanism designed to increase the network reliability and throughput. More information on reliability in wireless sensor networks can be found in [11, 12].

In WSN, attacks on transport protocols may be attacks on reliability and energy depleting attacks. Therefore, Buttyán and Grilo [2] proposed the SDTP protocol, which is a security extension of DTSN, based on the application of symmetric key cryptographic primitives. More details on SDTP can be found in Section 3.2.

Hash chains were used in the secure routing protocol Ariadne [13] and TESLA [14], and an example of securing protocols using Merkle-trees can be found in [15]. One of the first applications of a hash chain was proposed by Lamport [16], in which he suggested using hash chains as a password protection scheme in an insecure environment. This method was used in the S/Key one-time password system [17] developed for authentication in Unix-like operating systems. Other applications and hash chain optimization works can be found in [18, 19]. More details on hash chains can be found in Section 4.1.1.

Merkle-trees were originally proposed to efficiently handle many Lamport one-time signatures [20], where each packet can be signed by one Lamport key. An example of a practical application of Merkle-trees is the Google Wave protocol [21]. Below we detail the key features of DTSN, SDTP protocols, hash chains and Merkle-trees. More details on Merkle-trees can be found in Section 4.1.2

3 | ATTACKER MODEL AND SECURITY ISSUES

3.1 | Attacker Model

We assume a class of stealthy and internal attackers [22], whose purpose is to avoid detection of their activity. Our attacker has two main goals: to deceive the honest nodes that data packets have already been delivered while in reality they have been lost (which we refer to as a reliability attack), or to force the nodes to expend more energy than the amount actually needed (which we refer to as energy attack). In particular, we are interested in attacks where compromised nodes misbehave in more sophisticated ways, such that while causing huge damage, it is difficult to be discovered. “Brute force” type Denial-of-Service (DoS) attacks are not considered in this paper. In addition, we assume that the source and the destination of traffic flow (i.e., a path) are not compromised. The rationale is that one cannot do much against a misbehaving destination that acknowledges packets that it did not receive, or a source that deletes sent data packets before receiving any acknowledgments for them. We do assume, however, that any intermediate node may be compromised, and we want to eliminate its potential malicious effect on the system. Therefore, from this point on, our attacker is always stealthy and is part of the routing between the source and destination.

3.2 | Security issues in the SDTP protocol

The SDTP [2] protocol preserves the characteristics (and advantages) of DTSN, but extends it with the following cryptographic mechanisms. Each data packet with sequence number n is extended with two MAC (Message Authentication Code [23]) fields, an ACK MAC and a $NACK$ MAC. These two MACs are computed with two different keys, an ACK key (denoted by K^n_{ACK}) and a $NACK$ key (K^n_{NACK}), over the whole data packet (see Eq. 1). K^n_{ACK} and K^n_{NACK} are specific to the data packet with sequence number n; hence, they are referred to as per-packet keys [2].

Each K^n_{ACK} and K^n_{NACK} are computed by the source and the destination based on a one-way function over the *sequence number* n, the ACK or $NACK$ *session master key*, and the *constant* ACK or NACK, respectively. The ACK and $NACK$ session master keys are only known to the source and the destination, and are never revealed or sent out during the protocol.

$$\begin{aligned} \text{ACK MAC} &= MAC(Packet^n, K^n_{ACK}) \\ \text{NACK MAC} &= MAC(Packet^n, K^n_{NACK}) \end{aligned} \tag{1}$$

Upon receiving a data packet, by verifying the two MAC values, the destination node check the integrity and authenticity of data packet that the node received. In case the destination node receives EAR packet, it sends an ACK or a $NACK$ packet to the source according to the gaps in his received data buffer. Regarding the EAR, in the case where the destination concludes that there are no gaps, the destination reveals its ACK key (K^n_{ACK}, all the data packets with sequence number equal or less than n has been received), and sends it as part of the ACK packet.

In the case where the destination node decides that one data packet is missing (e.g, data packet with sequence number n), the destination node reveals its $NACK$ key (K^n_{NACK}) for each missing data packet and sends it as part of the $NACK$ packet with a bit map (indicates which data packets are missing). Any intermediate node checks the $NACK$ control packet and in the case of storing the corresponding packets, the intermediate node can verify the authentication of the control packet (ACK or $NACK$) with the included key. For each verification of the $NACK$ key, the intermediate node does the following: retransmits (if stored) the missing data packet, in the bit map it unsets the bit, from the $NACK$ keys it removes the corresponding key.

After each sending, the intermediate node check if the bitmap becomes clear. In case it is clear, the intermediate node sends an EAR message to the destination node and the $NACK$ packet is deleted and the intermediate node sends an ACK packet with the same ACK value as was in the $NACK$ packet. Moreover, both intermediate nodes and the source node maintain the largest verifiable acknowledged sequence number so far for each session, named as $MaxSN$, the aim of this value is to avoid replaying control packets.

Based on the fact that the session master keys are never leaked, and hence only the source node and the destination node can produce the right per-packet keys (ACK and $NACK$), the SDTP protocol is assumed to be secure [2]. As claimed by the authors in [2], the intermediate node can be sure that upon receiving ACK and $NACK$ packets are coming from the destination, since only the destination is able to reveal the correct keys. Moreover, due to the fact that the per-packet keys are computed by a one-way function, when the ACK and $NACK$ keys are revealed, an attacker can not reveal the master keys from them; hence, the yet unrevealed ACK and $NACK$ keys cannot be derived using the current keys. In the following, we discuss each of these attack modes.

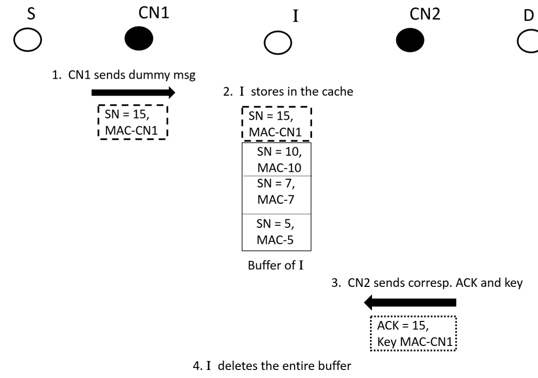

FIGURE 1 Creating a fake packet attack, where the sequence number of the fake (dummy) packet is 15.

Creating fake packets attack

This is an attack data packets can be eliminated from the intermediate nodes' caches by colluding attackers. Let us consider the following scenario, depicted in Fig. 1: Let define S, I, D be the source node, intermediate node, and destination node, respectively, and let $CN1$ and $CN2$ be the two cooperative compromised nodes between the source and the destination. We assume symmetrical links between $(S, CN1)$, $(CN1, I)$, $(I, CN2)$, and $(CN2, D)$ pairs as can be seen in Fig. 1. First, node $CN1$ creates a data packet m containing a MAC value computed with fake $NACK$ and ACK keys and sends the packet; then the intermediate node I stores the packet without being able to verify the MAC values. Later, the attacker $CN2$ generates fake ACK, $NACK$ packets with the corresponding keys (fake), those keys will match the MAC values of the stored m at node I generated by $C1$. Hence, node I considers these fake acknowledgment packets to be valid due to the verification success. Consequently, I eliminates the stored packets with a sequence number that is less than m (including m itself) from its cache, although some of the real packets have not been received by the destination, and updates its $MaxSN$ to be m. Intermediate nodes can be easily misled to believe that data packets have been delivered, although the destination has not received those packets. The worst scenario is when node $CN1$ is next to the source and $CN2$ is next to the destination and using the attack described above the entire chain between source and destination may delete packets although the destination did not receive them. Another basic reliability attack is when an attacker or attackers modify the data packets. Modifying the data leads to closing the session.

Forging $NACK$ packets attacks

These attacks, as depicted in Fig 2, give an attacker the ability to increase the overhead of the control packets and force multiple retransmissions. In general, this is not a regular attack that decreases the performance of the protocol/network. An attacker can use the bitmap and $NACK$ authentication values from one $NACK$ packet to generate a large number of valid $NACK$ packets with overlapping information. The attack is based on the following scenario: a source node sends a packet to a destination, and some of the intermediate nodes store this packet. Let us assume that for some reason (not a security reason) the packet does not reach the destination. After the source sends a data packet with the EAR flag set, the destination answers with a $NACK$ packet which includes a set of $NACK$ authentication values. An attacker can intercept the $NACK$ packet and generate other $\sum_{k=0 \dots n} \binom{n}{k} = 2^n$ real and useful $NACK$ packets from this NACK packet with overlapping information along with one ACK packet (without any $NACK$ authentication value), where n is the number of $NACK$ authentication values in the $NACK$ packet. The injection of the generated overlapping control packets will increase the control packet overhead, and as a result of the overlapping information the number of retransmissions of both data and control packets will increase. The control packet overhead can reach an upper bound of 2^{AW-1} real and useful packets (where AW is Acknowledgment Window [3]).

Note that some $NACK$s are useful and even the overlapping information in some cases (loss of some $NACK$s) can be advantageous. Therefore, instead of entirely preventing this attack, our purpose is to mitigate this attack by giving the intermediate node the ability to aggregate both new and overlapping information into one control packet.

Same authentication modification value in a $NACK$ packet attack

Here, an attacker can indirectly close a session using the bitmap included in the $NACK$ packet. The attack is based on the following scenario: a source node sends a packet to a destination, which is stored by some of the intermediate nodes. Unfortunately, for some reason (not a security reason) the packet does not reach destination. After the source sends a data packet with the EAR

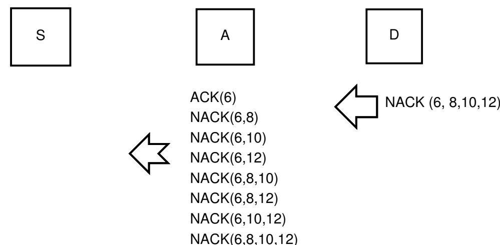

FIGURE 2 Injecting $\sum_{k=0...n} \binom{n}{k}$ $NACK$ Packets from a Single $NACK$ Packet. An example where a $NACK$ message can be transformed into ACK message.

flag set, the destination answers with a $NACK$ packet that includes a set of $NACK$ authentication values. The attacker always unsets the same bit in the bitmap, and erases the relevant $NACK$ authentication value from the forged $NACK$ packet. The $NACK$ packet propagates in the network with a lack of information about the missing packet until the $NACK$ packet becomes an ACK packet and reaches the source (an intermediate node cleared the entire bitmap). The lack of information about the missing packet will trigger the EAR timer in the source and after a while the EAR counter will reach the MAX value, which eventually leads to closing the session between the source and the destination [3].

In fact, to conceal malicious behavior and make the attack more difficult to detect, the attacker can modify the same $NACK$ authentication value as presented above, but only for short periods of time; this will not close the session but will increase the network delay.

Faking the session number attack

The attacker causes packet deletions from the intermediate node buffer by injecting packets with a new session number. In DTSN [3], by receiving a packet with a new session number the intermediate node deletes cache entries with the same source ID, destination ID, and application ID (for more information about the DTSN intermediate node algorithm see Fig. 3 in [3]). Therefore, by injecting a packet with the same source ID, destination ID, and application ID but with a different session number, an attacker can force an intermediate node to delete all the packets of the old (but real) session number.

Replaying or forging EAR flags attack

Here, by setting the EAR flag to a value 0, the attacker prevents the destination from sending control packets (ACK or $NACK$), whereas setting the EAR flag to 1 makes the destination send control packets unnecessarily. The attacker always sets the EAR bit to 0, with the result that the destination never receives a packet with EAR= 1, and hence, never sends an ACK or $NACK$ packet. On the other hand, the attacker can always set the EAR flag of packets to 1, making the destination send control pa‘ckets unnecessarily. Recall that the attacker can modify the EAR bit because it is not protected cryptographically.

Replaying $NACK$ packet attack

Here, the attacker replays old $NACK$ packets to force futile retransmissions of data packets.

Preventing retransmission attack

Here, the attacker attempts to prevent intermediate nodes from retransmitting packets by changing bitmap values in $NACK$ packets.

4 | THE STWSN PROTOCOL

STWSN is based on an efficient application of authentication values and asymmetric key crypto while enhancing the authentication and integrity protection of control packets. STWSN is tailored to the problem of authenticating and verifying the ACK and $NACK$ packets [1]. Our STWSN protocol is based on two main building blocks, hash chains and Merkle-trees and the general idea of STWSN is the following: two types of “per-packet" *authentication values* are used, ACK and $NACK$. Any intermediate node and the source node can verify the received ACK and $NACK$ packets by using the corresponding ACK and $NACK$ *authentication values*, respectively. Each ACK authentication value is an element of a hash chain [5], whereas a

$NACK$ authentication value is composed of a leaf and its corresponding sibling nodes along the path from the leaf to the root in a Merkle-tree [6]. The computation of the per-packet keys $K_{ACK}^{(n)}$ and $K_{NACK}^{(n)}$ is based on the application of hash chains and Merkle-trees. We adopt the notation and notion master secrets K_{ACK}; K_{NACK} from SDTP [2]; i.e., they are computed in the same way as in the case of SDTP, based on a pre-shared secret.

In the following, we described the ACK and $NACK$ authentication values in detail and then explain for each possible node (source, destination and intermediate) the main algorithm based on the security mechanisms to mitigate and prevent energy and reliability attacks.

4.1 | Preliminary

4.1.1 | The ACK Authentication Values

In order to produce many one-time keys from a single key we can use a hash-chain [5,24]. Lets define x as the initial value, the hash function as h, and the hash chain initial value as $v_m = h(x)$. Then, according to the above definitions the i-th element of the hash chain v_i will be computed as $v_i = h(v_{i+1}) = h^{(m-i)}(v_m)$. The important property of the hash chain, one-way property, is that the elements can be computed in one direction easily, however, not in the reverse direction. In other words, if an attacker knows v_i, might compute any $v_j = h^{(i-j)}(v_i)$ for any $j < i$, but the attacker can not compute any v_k for $k > i$. Therefore, in a cost of a single digital signature and at the cost of the computation and storage of the hash chain we can use hash chain for repeated authentications [24].

Therefore, anyone that wants to authenticate itself should compute a hash chain $v_m, v_{m-1}, \dots, v_0$ of length $m+1$, and digitally sign the last element v_0 (termed the root of the hash). The digital signature can be verified by anyone using the public signature verification key of the entity. Later on, by revealing in reverse order the elements of the hash chain the entity can authenticate itself repeatedly (at most m times). More precisely, at the i-th authentication, the entity reveals v_i. In order to verify, the verifier can use one of the following ways: remember the last used hash chain element v_{i-1}, and verify v_i with a single hash computation or hash this value i times and check whether the result matches v_0 that has been signed by the entity.

As explain above, we used hash chain elements and accept them only once. Therefore, in case an element v_i is accepted, the other elements in the hash chain $v_{i-1}, \dots, v_1$ can no longer be used. As we explain above about hash chain (the one-way property), the elements in the hash chain $v_{i+1}, \dots, v_m$ that can still be used for authentication cannot be computed by anybody except the entity that knows v_m.

In any case the verifier sees hash chain elements $v_{i+1}, \dots, v_m$, it can be sure that those elements have been revealed by the entity with its signature [24]. Therefore, the i-th element of a hash chain is the authentication value associated with the ACK packet referring to the i-th data packet. The source node, at the beginning of each session, generates the ACK master secret K_{ACK} and uses the initial value $K_{ACK}^{(m)} = h(K_{ACK})$ to calculate a hash chain of size $m+1$, where m is the number of data packets that the source node needs to send in the session to the destination node. Each element of the calculated hash chain represents a *per packet ACK authentication value* as follows: $K_{ACK}^{(m)}, K_{ACK}^{(m-1)}, \dots, K_{ACK}^{(1)}, K_{ACK}^{(0)}$,
where $K_{ACK}^{(i)} = h(K_{ACK}^{(i+1)})$ and h is a one-way hash function where $K_{ACK}^{(i)}$ represents the ACK authentication value corresponding to the data packet with sequence number i and the value $K_{ACK}^{(0)}$ is the root of the hash chain. Finally, in order to stored the hash chain efficiently a some storage complexity that is logarithmic in the length m we can use some of the techniques from [5,18,19].

4.1.2 | The $NACK$ Authentication Values

As explained above, hash chains can produce many one-time keys from a single key. However, a hash-chain has several limitations, one of which is that the elements can only be revealed sequentially (one after the other). Using a hash chain to authenticate and verify the $NACK$ messages has a major drawback. By revealing element v_i any attacker can reveal $v_{i-1}, ..., v_0$ elements and use each of them to request a retransmission of a message that has already been received by the destination. One of the ways to overcome this problem is to use Merkle-trees [6,24]. The operation of a Merkle-tree can be defined as follows: Let define $v_1, v_2, \dots, v_{2^\ell}$ as the set of values that we want to authenticate. First, each value v_i we hash into v_i' using a one-way hash function. Second, the hashed values become the leaves of a binary tree. Next, to each internal vertex in the binary tree u, we assign a value that is computed as the hash of the values assigned to the two children of u [24]. Finally, we take the root of the tree and digitally sign its value.

Later, by revealing v_i and all the relevant values assigned to the siblings of the vertices on the path in the Merkle tree from v_i' to the root we can authenticate the value v_i. By hashing these values and compare the result to the value assigned to the root,

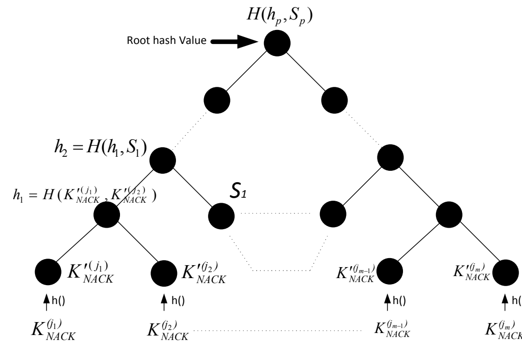

FIGURE 3 The STWSN Merkle-tree structure [7].

the verifier can check if the two values match. If so, the value v_i is accepted by the verifier as authentic. Note, we cannot use the revealed value v_i and the values assigned to the siblings to compute an as yet unrevealed value v_j. This ensures that the values can be revealed in any order.

In case of both hash-chains and Merkle-trees, when a digital signature is used, the public signature verification key has to be known by every node. Since sensor node digital signing and verify procedures are costly, it is useful to implement the signing procedure for bootstrapping alone. The signing procedure is only important for secure distribution of the root of the hash chain and the Merkle-tree root.

In our protocl (STWSN), the hash chain is not applicable due to the fact that several $NACK$ authentication values are revealed at a time in any order. Therefore, we need to search for another security building block. Based on section **??** we use the binary Merkle-tree to authenticate the $NACK$ packets.

In the first step, we need to compute the so-called *NACK secret values* based on a Pseudo-Random Function (PRF) as follows:

$$K^{(n)}_{NACK} = PRF(K_{NACK}, \text{“}\textit{per packet NACK secret}\text{”}, n),$$

where n is a sequence number of a data packet.

In the second step, we hash each *NACK secret value*. In the third step, we assign the hashed values to the leaves of the Merkle-tree: $K'^{(n)}_{NACK} = h(K^{(n)}_{NACK})$. We refer to these leaf values as *NACK leaf values*. Based on $NACK$ leaf values, the internal nodes of the Merkle-tree are computed. The STWSN Merkle-tree structure can be found in Fig. 3.

4.2 | STWSN – Source Mechanism

When the source wants to open a session, it first computes the following: (1) the session master secret K; (2) the ACK master secret K_{ACK}; (3) the $NACK$ master secret K_{NACK}. Then, the source calculates the $NACK$ authentication values and ACK authentication values using the knowledge about the number of data packets in the session and the master secrets. Thereafter, a Merkle-tree and a hash chain are generated for the session, based on the calculated parameters.

After the calculations, the source sends an open session packet (see Fig. 4) with the following: the root of the hash chain ($K^{(0)}_{ACK}$), the hash chain length ($m + 1$), the number of Merkle-tree roots (t) in the case where we used several tress (see Section 10.2 for more on the advantages and disadvantages of using several Merkle-trees), the Session identifier $SessionID$, the source node ID, the destination node ID, and the root values of the t Merkle-trees (for simplicity we chose $t = 1$). Finally, the source digitally signs the whole packet (e.g., ECC [25]).

Since the open session packet may not reach the destination, the source node may need to retransmit the packet again (open-session packet). Therefore, after sending a new open session packet, the source node initiates an open-session timer, and upon timeout (the time has elapsed) without any feedback (ACK message) from the destination as to the successful receipt of the open session packet, the source node retransmits the open session packet again. The source node also limits the number of retransmissions of the open session packet.

When the source node receives an ACK packet from the destination node, the source verifies the ACK packet and starts sending data packets to the destination node. Each data packet is extended with a MAC, computed over the whole data packet except for the flags in the header of the data packet using the shared secret with the destination. The MAC over the flags (e.g.

Hash Chain Root	Hash Length	# Merkle roots	Session Identifier
Source, Destination			
Merkle Tree Roots			
Authentication Data			

FIGURE 4 Open Session packet

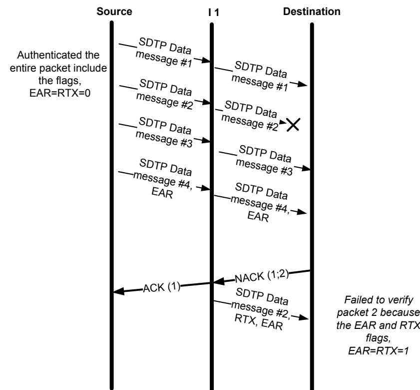

FIGURE 5 SDTP scenario where the MAC value is computed over the packet and the flags; for simplicity we reproduce the scenario in [3] (Fig. 4 in [3]).

EAR and RTX) in the header is not computed because the EAR and RTX flags may be modified by the intermediate nodes during the protocol. An example of a case where the flags are legitimately modified by the intermediate nodes can be seen in Fig. 5. It shows that EAR and RTX flags should not be included in the MAC since this may lead to verification failure.

In this scenario, packet #2, in which $EAR = RTX = 0$, is lost. When the destination node receives packet #4 which piggy-backed an EAR packet, it sends a $NACK$ packet requiring the retransmission of packet #2. However, when node 1 (I1) receives the $NACK$ packet it retransmits packet #2 and sets the EAR and RTX flags without recalculating the MAC value. At the end, verification at the destination node of the MAC corresponding to packet #2 fails since the MAC was originally computed over the zero flag values. Unfortunately, due to the fact that we not authenticating the flags, an attacker can modify the flags. This will increase the overhead caused by the operations that nodes perform due to the fake values. In Section 5.3.2 we discuss the solution to this problem.

Upon receiving an ACK packet (which includes the ACK authentication value, $K^{(i)}_{ACK}$) corresponding to the data packet of sequence number i, it iteratively hashes the ACK authentication value i times. If it is equal to the $K^{(0)}_{ACK}$ (root hash value), the ACK packet is accepted and the source node eliminates packets with sequence numbers smaller than or equal to i from its cache. Then, the source node updates its value of $MaxSN$ i; otherwise, it ignores and drops the ACK packet. When receiving a $NACK$ packet that includes the ACK authentication value $K^{(i)}_{ACK}$, $NACK$ authentication values (secret values $K^{(i+1)}_{NACK}, \ldots,$ $K^{(i+j)}_{NACK}$, and their corresponding sibling values), the source node first checks the ACK authentication value and performs the same steps as explained above for ACK authentication. Then, the source node continues verifying the $NACK$ authentication values. For each set bit in the bitmap, the source node verifies the $NACK$ authentication values and upon success, retransmits the required packets. Algorithm 1 in Section 9.1 depicts the source node algorithm.

4.3 | STWSN – Destination Mechanism

Upon receiving an open-session packet from the source, it verifies the signature computed on the packet by comparing the MAC value from the open session packet to the output of the MAC function over the receiving packet. Upon success, the destination node does the following: (1) generate the session master secret; (2) generate the ACK master secret; (3) generate the $NACK$ master secret; (4) generate the hash chain (5) generate the Merkle-tree (in case $t = 1$). Finally, the destination reveals the ACK key corresponding to the open-session packet; namely, the ACK key for the packet with sequence number zero (the open session has sequence number zero). Then the destination node sends the ACK packet with the parameter related to the open session packet to the source node. As explained in [2], the easiest way to generate the session master key is to derive it from a pre-established shared secret value; otherwise, in a more general setting, use an authenticated Diffie-Hellman protocol variant [26].

When the destination node receives a data packet with sequence number i, it does the following: (1) the destination using the secret shared between the source node and the destination node to check the authentication data field; (2) Upon success, received packet delivered to the upper layer; otherwise, the data packet is ignored and dropped.

If the data packet has a set EAR flag, the destination sends an ACK or a $NACK$ packet depending on the gaps in the received data packet stream. The structure of the ACK packets is extended with an ACK authentication value field. Similarly, the $NACK$ packet is extended with the ACK authentication value (with base sequence number i, $K_{ACK}^{(i)}$), because the semantics of the base sequence number in the $NACK$ packets is the same as that of the sequence number in the ACK packets. In addition, if the j-th bit is set in the bitmap, the $NACK$ packet is also extended with $K_{NACK}^{(i+j)}$ (the $NACK$ secret value) and its sibling authentication values. Therefore, $NACK$ packets is extended with an ACK authentication value field and a variable number of $NACK$ secret and sibling value fields.

In the DTSN [3] and SDTP [2] protocol, the destination sends an ACK or $NACK$ packet upon receipt of an EAR. However, this may lead to EAR replay or EAR forging attacks where the EAR flag is set/unset by an attacker(s). Therefore, to mitigate the effect of these attacks, STWSN uses two new mechanisms: (1) limiting the number of responses to EARs; (2) a status timer (dynamic or static). In a finite period of time (destination EAR timer) each destination node should not send more than X control packets (ACK, $NACK$), where the X value can be dynamic or static. In the case where we choose a high X value, it will increase the overhead of the control packet, whereas if we choose a low X value it may be considered under the condition that the duration between two status (triggered or created[1]) packets is less than the source node EAR timer.

The status timer is set at the destination node. Upon time out, the destination node will automatically send an updated ACK or $NACK$ packet. The status timer duration can be a function of the source EAR timer. Moreover, the destination node limits the number of responses upon receiving a set EAR flag. Algorithm 2 in Section 9.2 depicts the destination node algorithm.

4.4 | STWSN – Intermediate Node Mechanism

Upon receiving an open session packet, the intermediate node verifies the signatures computed on the packet. Upon success, the intermediate node stores the following regarding this session: (1) the hash chain root value; (2) the tree root values; (3) the $SessionID$ included in the open session packet. Then, the intermediate node forwards towards the destination the open session packet. In case the verification failed, the intermediate node will not store packets in the current session (changes its probability to store packets to zero). Upon receipt of the corresponding ACK packet related to the open session parameters stored in the intermediate node, it uses the hash chain root stored value to verify the corresponding ACK key and in case of success will start forwarding data and control packets for this session.

Upon receipt of a data packet of an already opened session, an intermediate node stores the data packet with probability p and (always) forwards the data packet towards the destination node.

Note that the intermediate node follows the same steps in the case of receiving ACK or $NACK$ control packets as the source node. When an ACK packet that refers to the packet with sequence number i is received by an intermediate node, the intermediate node hashes the ACK authentication value i times, and compares the result to the stored root hash chain value to verify the correctness of the ACK authentication value. If the two values are equal (i.e., $K_{ACK}^{(0)} = h^i(K_{ACK}^{(i)})$), all the stored packets with a sequence number less than i are deleted, and the intermediate node updates its $MaxSN$ value to i. Afterward, the intermediate node transmits the ACK packet to the next intermediate node towards the source node. In case the verification fails, the ACK packet is ignored and dropped. If $i \leq MaxSN$ the ACK packet is also ignored and dropped.

[1]Triggered refers to control packets sent after receiving the EAR packet; created refers to control packets sent after the status timer has expired.

In the case of receiving a $NACK$ packet with base sequence number i, the intermediate node compares the base sequence number with its $MaxSN$ value. If i is smaller than or equal to $MaxSN$, the $NACK$ packet may still contain useful information in the bitmap. Regarding the bitmap, if the j-th bit is set and $i + j > MaxSN$, the intermediate node first verifies the corresponding $NACK$ authentication value and upon success retransmits the corresponding data packet stored in its cache which needs to be resent. However, if the j-th bit is set and $i + j \leq MaxSN$, the intermediate node clears the bit and removes the corresponding $NACK$ authentication value from the $NACK$ packet. Afterward, the $NACK$ packet or the ACK packet (if the bit map is no longer set) transmits to the next intermediate node.

Moreover, if any of the data packets that correspond to the set bits in the bitmap of the $NACK$ packet are stored in the cache of the intermediate node, for each data packet, it verifies the $NACK$ authentication value corresponding to the data packet. In particular, the intermediate node calculates the root and compares it to the Merkle-tree root cached by the intermediate node based on the $NACK$ authentication value. If the two values are equal, the intermediate node does the following: (1) the data packet is scheduled for retransmission; (2) the corresponding bit in the $NACK$ packet is cleared; (3) the $NACK$ authentication value is removed from the $NACK$ packet.

In the case where the bitmap is cleared, the intermediate node sends an ACK packet with n (the ACK value of the $NACK$ message, see Fig. 2 for example) the same sequence number as the $NACK$ packet, and sets the EAR flag in the last retransmitted data packet. If the bitmap is not cleared, the $NACK$ packet is transmitted to the next intermediate node towards the source. In the case where the intermediate node does not have any of these data packets, it passes on the $NACK$ without modification.

Because we want to mitigate several attacks, the intermediate node adds the following mechanisms to the actions above: an intermediate node does not immediately forward control packets, after successfully verifying the $ACK/NACK$ packets (except the $ACK/NACK$ of an open session message) but rather only after a certain length of time. Specifically, intermediate nodes set an aggregate-timer, which can be either static or dynamic. If within this period of time more than a certain number of control packets (ACK, $NACK$) are received by the intermediate node, it tries to merge the verification information into one control packet (i.e., a form of aggregation) from the received control packets; otherwise, the intermediate node will send the original $ACK/NACK$ packets without any changes. The intermediate node should correlate is sum of the aggregate timers to the source EAR timer. Hence, the timer value may be a function of several parmeters such as the source EAR timer and the maximum number of nodes in the path.

The intermediate node is also using a timer called the retransmission-timer (denoted by Re_timer_i), in order to limit the transmission rate of a retransmitted data packet with sequence number i. Further, the number of retransmissions ($Retransmission_i$) per stored data packet with sequence number i is limited as well. After a certain number of retransmissions of the same data packet, although the intermediate node it stored the data packet it only forwards the $NACK$ packet without retransmission. However, in the worst case, an intermediate node still can be made to retransmit unnecessarily up to the limit.

When receiving a certain number T of ACK packets with the same acknowledge value equal to $MaxSN$, the intermediate node automatically retransmits the first packet in its buffer that has a sequence number greater than $MaxSN$ and resets the count. In addition, these "multiple" ACK control packets are forwarded towards the source. An illustration of this scenario can be found in Fig. 6. In this scenario, S and D are the source and destination, node 1 is an intermediate honest node, and node A is the attacker node. The destination sends a $NACK$ packet with the ACK value 7 and the bitmap in which the bits corresponding to packets 8 and 9 are set. The attacker intercepts this $NACK$ packet and deletes both the bit referring to packet 9 and the corresponding $NACK$ authentication value (getting bit map $NACK(7, 8)$). Assume that packet 8 has been retransmitted and arrived at destination. After a while, the destination acknowledges the reception of packet 8 and requires the retransmission of packet 9. However, the attacker intercepts this $NACK$, deletes the bit referring to packet 9 and the corresponding $NACK$ authentication value, thus getting bit map $ACK(8)$, and then forwards it towards the source. Moreover, the attacker repeats these steps whenever the $NACK(8, 9)$ is sent by the destination. Node 1, after receiving the ACK with the same ACK value that equals the $MaxSN$ three times ($T = 3$), automatically retransmits its first stored packet that has a sequence number greater than 8 (which is packet 9). Thus, our proposed mechanism ensures that although the attacker tries to prevent this by always deleting the corresponding bit, packet 9 will be retransmitted. Note that the source has a timer for packet 9 so in the case where the attacker eliminates the retransmission, after a while the source may recognize that there is an attacker (several timeouts for the same packet). Unfortunately, we can only mitigate the impact of the above attack; however, the period of time for this kind of attack is relatively short, because each node has an aggregate timer and the source will filter the irrelevant control packets (as

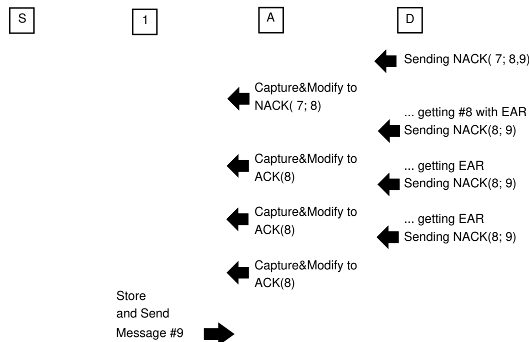

FIGURE 6 Special case of the modification of the same $NACK$ Packet, $T = 3$.

explained in Section 3.2). Moreover, setting a low T value will increase the number of unnecessary ACK packets received by the source but decrease the reaction time of the network to deal with the above attack[2].

In the case where an intermediate node has no room in his buffer and needs to delete a packet, before deleting the first packet in the buffer it will send with probability q the first packet in its buffer that has a sequence number greater than the intermediate node $MaxSN$. Note that, q may be a different probability from the caching probability. Only then does the intermediate node delete this packet. However, the probability of q needs to be set close to 0 in case the attacker is able to inject fake packets with a high sequence number, which can cause a retransmission of a fake packet. Since injecting fake packets has limited effectiveness on the network (as explained in Section 3.2), q can be larger than 0.5.

For detailed processing of the intermediate node algorithm see Fig. 14 in Section 9.3.

5 | SECURITY ANALYSIS

In what follows, we analyze the security of STWSN based on a formal language called $crypt_{time}^{prob}$ calculus [7, 8]. We formally prove that STWSN prevents all the discussed reliability attacks and mitigates the impact of energy depletion attacks to which both DTSN and SDTP are vulnerable. We note that this paper provide much more proofs for more security properties compared to [7, 8].

5.1 | The $crypt_{time}^{prob}$ calculus

We only provide a very brief overview of $crypt_{time}^{prob}$ calculus for the reader to understand the proofs; interested readers are referred to our report [8] for further details. $crypt_{time}^{prob}$ is a probabilistic timed calculus, for reasoning about cryptographic protocols that include timers and probabilistic operation.

Syntax: The formal syntax of $crypt_{time}^{prob}$ is composed of *terms*, *probabilistic timed processes*, and *equations*.

- Terms (denoted by T) involve secret keys, encryption, hash and MAC functions, digital signatures computed over certain messages, as well as their composition. *Terms* also includes communication channels defined between nodes, which facilitates message exchange among communication partners.
- The set of *probabilistic timed processes* defines the internal operation (sequence of defined actions) of the nodes. Processes are denoted by *procName*, where *Name* can be the name of the process (e.g., *procSrc* is a process that defines the operation of the source). A transport protocol can be defined by the (parallel) composition of processes in which each process specifies the behavior of each node.
- Equations express the equality of two terms ($T1 = T2$), and are used to model the verification of cryptographic functions.

Semantics: The semantics of $crypt_{time}^{prob}$ is based on a probabilistic timed labelled transition system (PTTS), which consists of a set of labelled transition relations of the form $st_1 \overset{\alpha,\, d}{\rightarrow} st_2$, where st_1 is the current state of a given process, and st_2 is the state

[2]Intuitively, this solution is similar to the TCP Fast Retransmission mechanism.

after an action α has been executed, consuming d time units. Each state st is composed of the process and the clock valuation that holds at that state, namely, $st = (procName, v)$.

The *weak probabilistic timed bisimilarity (prob-timed bisimilarity)* relation is defined to prove or refute the similarity (equivalence) between the ideal and the real operations of a WSN transport protocol.

Definition 1. (**Weak prob-timed labeled bisimilarity**)
We say that two states $st_1 = (procName1, v_1)$ and $st_2 = (procName2, v_2)$ are weak prob-timed labeled bisimilar, denoted by $(st_1 \; \mathfrak{R}_t^p \; st_2)$ iff

1. Static equivalence: An external attacker who can listen to and intercept the communication on the entire network cannot distinguish the packet output or input in the states st_1 and st_2;
2. if state st'_1 can be reached from st_1 after a silent (invisible) action after a d_1 time unit, then there exists a state st'_2 that be reached via a corresponding silent action trace/transition after a d_2 time unit from st_2, such that $st'_1 \; \mathfrak{R}_t^p \; st'_2$ holds again. Silent actions are action invisible to the external observer (e.g., signature/MAC verification, decryption, etc.).
3. If state st'_1 can be reached from st_1 after a non-silent (visible) labeled transition (i.e., data sending or receiving) after d_1 time units, then there exists a state st'_2 that be reached via a corresponding non-silent action trace/transition after a d_2 time unit from st_2, such that $st'_1 \; \mathfrak{R}_t^p \; st'_2$ holds again.

and vice versa.

The attacker model: We assume that an attacker can eavesdrop on and catch the messages output by the honest nodes, and can modify the elements of any captured packets, including control packets, the EAR and RTX bits, as well as the sequence numbers in data packets. The attacker can also compose whole data or control packets containing the elements it has, as well as send and forward packets.

5.2 | Security proof technique based on the PTTS

For each attack scenario, we define a process that captures an ideal operation of the protocol that is not vulnerable to that attack. For example, the ideal operation requires that the honest communication partners are always aware of the correct packet they should receive from other nodes, regardless of what the attackers do, or the ideal version always limits the number of control/acknowledgement packets. If the ideal and real protocol processes are weak prob-timed bisimilar (with the same attacker model), then the real protocol is not vulnerable to that attack.

Definition 2. *Let the* $crypt_{time}^{prob}$ *processes procProto* and $procProto^{ideal}$ specify the real and ideal versions of some protocol *Proto*, respectively.

- **Security Proof:** We say that *Proto* is secure (up to the ideal specification) if $(procProto, v^{init})$ and $(procProto^{ideal}, v^{init}_{ideal})$ are weak prob-timed bisimilar:

$$(procProto, v^{init}) \approx_{pt} (procProto^{ideal}, v^{init}_{ideal}),$$

- **Refute:** Let $procProto^{noSec}$ be a process that models a protocol *Proto* without a security mechanism *Sec*, and *procProto* is the counterpart with *Sec* implemented in it. Assume that $procProto^{noSec}$ is vulnerable to an attack *Att* modelled by a sequence of labelled transition LT_{Att}. We say that *Proto* is more secure against *Att* than $Proto^{noSec}$ if $(procProto, v^{init})$ and $(procProto^{noSec}, v^{init})$ are **not** weak prob-timed bisimilar; namely

$$(procProto, v^{init}) \approx_{pt} (procProto^{noSec}, v^{init})$$

does not hold due to LT_{Att}, where v^{init} and v^{init}_{ideal} are the initial values of the clocks.

Intuitively, Definition 2 means that *Proto* is secure if by observing the packets sent and received by nodes, the attacker is not be able to differentiate between the operation of the two protocol instances.

5.3 | Formal security analysis of STWSN

As mentioned above, terms (T) can be used to model packets elements, including cryptographic primitives, starting from secret and public keys, to hash and MAC functions. Equations are defined for modelling cryptographic verifications:

$$T = n \mid K_{ack} \mid K_{nack} \mid K_{sd} \mid SK_{src} \mid ACK \mid NACK \mid K(n, ACK) \mid K(n, NACK) \mid sign(T, SK_{src}) \\ \mid H(T) \mid mac(T, K(n, ACK)) \mid mac(T, K(n, NACK)) \mid ok \mid (T_1, \ldots, T_m);$$

Equations:

$checkmac(mac(T, K(n, ACK)), K(n, ACK)) = ok;$
$checkmac(mac(T, K(n, NACK)), K(n, NACK)) = ok,$
$H(T_1) = H(T_2)$, iff $T_1 = T_2$.

K_{ack}, K_{nack}, and K_{sd} are the ACK and $NACK$ master keys, as well as the shared key of the source and the destination for a session. n represent any constant data including sequence numbers, ACK authentication values, and EAR/RTX bits, etc. The functions $sign(T, SK_{src})$, $mac(T, K(n, ACK))$, and $H(T)$ define a digital signature computed on packet T using the secret key SK_{src}, a MAC and an one-way hash computed on T, respectively. The equation $H(T_1) = H(T_2)$ if and only if T_1 and T_2 are the same. The equation $checkmac(mac(t, K(n, ACK)), K(n, ACK)) = ok$ defines MAC verification, using the corresponding public key ACK key $K(n, ACK)$).

5.3.1 | Analyzing the resistance to reliability attacks

As explained in Section 3.2, the two basic attacks on reliability take the form of forging ACK and $NACK$ packets. Assuming that an attacker has intercepted an ACK or $NACK$ packet in which the ACK value is n. By changing the correct ACK value n to a larger m, the attacker attempts to delete packets from the buffers although they have not yet been acknowledged by the destination.

Proposition 1. The STWSN protocol is secure to attempts to forge ACK and $NACK$ packets.

To perform a successful attack by increasing the ack value n_1 to some greater n_2 in ACK packets, the attacker has to include a correct ACK authentication value $K^{(n_2)}_{ACK}$, which is difficult. This is because up to this point, only $K^{(0)}_{ACK}$,..., $K^{(n_1)}_{ACK}$ have been revealed by the destination. However, computing $K^{(n_2)}_{ACK}$ from $K^{(0)}_{ACK}$,...,$K^{(n_1)}_{ACK}$ is hard because of the one-way property of the hash function.

Proof. Formally, this is proven by the fact that there is no equation defined for $H(T)$ (since hash functions are one-way), there is no way for the attacker to violate the bisimilarity

$$(procSTWSN, v^{init}) \approx_{pt} (procSTWSN^{ideal}, v^{init}_{ideal}),$$

We defined the ideal and real versions of STWSN as $crypt^{prob}_{time}$ processes, and we showed that every labeled transition performed by the ideal version can be simulated by a corresponding transition trace in the real version, and vice versa. □

Consider now the case of a $NACK$ packet forgery where valid $NACK$ packets is created by the attacker, either a whole $NACK$ packet or forging some bits of the bitmap of valid packers. To carry out an attack, a valid $NACK$ authentication value needs to be added into a $NACK$ packet. Hence, the attacker must be able to compute the secret values based solely on the upper-level hash values (or the siblings). However, this is hard due to the one-way property of the hash function used in the Merkle trees.

Proof. Similarly, the formal proof relies on the one-way property of the hash function, and hence, the infeasibility of computing the required nodes of the Merkle tree. Again, formally, this is proven by showing that every labeled transition performed by the ideal version of the system can be simulated by a corresponding transition trace in the real version, and vice versa. This is because there is not any equation defined for function $H(T)$, and hence neither the ideal nor the real systems can perform a labelled transition trace that extracts T from $H(T)$. Therefore, there is no way for the attacker to violate the bisimilarity

$$(procSTWSN, v^{init}) \approx_{pt} (procSTWSN^{ideal}, v^{init}_{ideal}).$$

□

Another basic attack on reliability is when an attacker (attackers) modifies the data packets, which results in closing the session.

Proposition 2. The STWSN protocol is secure to attempts to modifying data packets.

Data packets are secured with MAC [23]; thus, a modification to the data part will be detected at the destination. However, attackers can cooperate to bypass the protection provided by MAC.

Proof. Again, this is proven by showing that every labeled transition performed by the ideal version of the system can be simulated by a corresponding transition trace in the real version, and vice versa. This is due to the fact that this time either there is

1. no labelled transition for computing the MACs $mac(T, K(n, ACK))$ and $mac(T, K(n, NACK))$ or
2. no labelled transition for MAC verifications $checkmac(mac(T, K(n, ACK)), K(n, ACK))= ok$, $checkmac(mac(T, K(n, NACK)), K(n, NACK))= ok$ (for any T) can be launched in either the ideal or real systems, because the attackers do not possess the keys $K(n, ACK)$ and $K(n, NACK)$.

Consequently, in this manner, the attackers cannot violate the bisimilarity

$$(procSTWSN, v^{init}) \approx_{pt} (procSTWSN^{ideal}, v^{init}_{ideal}).$$

□

Proposition 3. The STWSN protocol is secure to attempts to create fake packet attacks.

Recalling the *creating fake packets* attack presented in Section 3.2, which is one critical vulnerability of SDTP, that can enable the attackers to force intermediate nodes to delete undelivered packets in their buffers. The feasibility of this attack relies on the fact that the authenticity of the keys used for checking the authenticity of $ACK/NACK$ packets are not verified; making it possible to fake MACs with self-created keys.

The attack cannot be carried out in STWSN as the authentication values are parts of the Merkle tree and hash chains, and due to the one-way property of hash functions, computing an undisclosed ACK authentication value of packet n_2 ($K^{(n_2)}_{ACK}$) from the already revealed $K^{(n_1)}_{ACK}$ ($n_2 > n_1$) is difficult. Computation of the undisclosed NACK authentication values from the already revealed ones is hard for the same reason. Further, the attacker(s) will not be able to send to the honest nodes any fake self-created Merkle trees or hash chain since they cannot forge the digital signature of the source.

Proof. The formal proof of this proposition is similar to the case of Proposition 1. □

5.3.2 | Formal analysis of the resistance to energy depleting attacks

In what follows, we formally analyze how STWSN mitigates the impact of energy depleting attacks. Since the EAR flags are not appropriated secured in the STWSN packets, its modification cannot be detected. This may serve as a starting point for a possible attack, called the EAR *modification* attack, in which the attacker can set or unset the EAR flag in any captured packet.

Proposition 4. The STWSN protocol mitigates the effect of an EAR modification attack.

Mitigating the impact of the attack prevents attackers from critically depleting the energy of the nodes. First, by setting the EAR flag (EAR flag=1), the attacker increases the control packet overhead by forcing the destination to send more, unnecessary $ACK/NACK$ packets. As described in subsection 4.3, the impact of this attack is alleviated because once the destination receives a set EAR flag it only considers a limited number of responses. In a finite time period (set by the destination EAR timer) the number of control packets sent by the destination will not exceed a certain number X[3]. By unsetting the EAR flag (EAR flag=0), the attacker could achieve that no control packets will arrive at the source. However, this cannot happen for a long time as an EAR timer is launced by the source. Once the EAR timer expires, a new EAR packet will be retransmit by the source. In addition, as described in Section 4.4, in STWSN, with the status timer and the constraint set on the number of control packets sent by the destination in a given time, we can mitigate the overhead posed by the attack.

[3] The value of X can be either dynamic or static

Proof. We assume that X is the number for which no error will be sent by the destination node due to energy depletion caused by futile sending of control packets. We let the constant LOWBATTERY represent the signal that destination D will send out on channel c_{signal} once a low battery (under a certain threshold) is detected. Further, let $STWSN^{noX}$ be the $STWSN$ protocol without the limit$X$. Then, we prove that the bisimilarity

$$(procSTWSN^{noX}, v^{init}) \approx_{pt} (procSTWSN, v^{init}).$$

does not hold, because in $procSTWSN^{noX}$ there exists a Dst_EAR_Timer value and a corresponding labelled trace ends with the transition $\xrightarrow{\overline{c_{signal}}\langle LOWBATTERY\rangle, d}$, while this is not the case in $procSTWSN$; hence, process $procSTWSN$ cannot simulate $procSTWSN^{noX}$.

□

Proposition 5. The STWSN protocol mitigates the effect of forging a $NACK$ packet attack.

Note that $NACK$s with overlapping information may occur in some normal cases; hence, instead of entirely preventing this attack, our purpose is to mitigate this attack by giving the intermediate node the ability to aggregate both new and overlapping information into one control packet. Doing this will reduce the overhead of intermediate/source nodes needed to handle each overlapping $NACK$s separately.

Upon receiving a control packet, an intermediate node waits for a short time; assume that during this period of time a new control packet arrives. If the information is new (i.e., ACK with a higher value or $NACK$ with a new bitmap) it is useful to aggregate the information into one control packet; if there is some overlap then it is also useful to aggregate the information. Note that control packets can be aggregated only after a successful verification. This is the same technique as delayed ACK in TCP.

Moreover, the use of a retransmission-timer at an intermediate node also helps mitigate the impact of replay attacks in which the attacker uses the same $NACK$ packets (with the same bitmap) or forging $NACK$ packets (with the bitmap that was created in previous $NACK$ packets); see Section 4.4 for a detailed discussion regarding the overhead. Therefore, these solutions are successful in mitigating the effect of forging $NACK$ packets.

Proof. Again, we let the constant LOWBATTERY representing the signal that an intermediate node I will send out on channel c_{signal} once a low battery is detected. Further, let $STWSN^{noAgg}$ be the $STWSN$ protocol without an aggregation procedure. We prove that the bisimilarity

$$(procSTWSN^{noAgg}, v^{init}) \approx_{pt} (procSTWSN, v^{init}).$$

does not hold, because in $procSTWSN^{noAgg}$ there exists a Agg_Timer value and a corresponding labelled trace that ends with $\xrightarrow{\overline{c_{signal}}\langle LOWBATTERY\rangle, d}$, while this is not the case in $procSTWSN$; hence, process $procSTWSN$ cannot simulate $procSTWSN^{noAgg}$.
□

Proposition 6. The STWSN protocol mitigates the effect of modification of the same authentication value in a $NACK$ packet attack.

As described in Section 4.4, by automatically sending and forwarding a data packet after receiving T pieces of the same ACK packets, the probability that a data packet will be received by the destination is increased. Moreover, this can increase the effectiveness of the network where an intermediate node sends a packet, right before deleting it, whose sequence number is greater than its $MaxSN$ (which means that the node did not receive any ACK for this packet). Both methods can increase the probability that a packet will be received by the destination and mitigate the modification of the same authentication value in a $NACK$ packet attack.

Proof. We let $STWSN^{noT}$ be the $STWSN$ protocol without the limit T for the same ACK packet. We prove that the bisimilarity

$$(procSTWSN^{noT}, v^{init}) \approx_{pt} (procSTWSN, v^{init}).$$

does not hold, because in $procSTWSN$ there exists a labelled trace that ends with the transition $\xrightarrow{\overline{c_{si}}\langle Pck\rangle, d}$ after the source receives the same ACK T times on channel c_{SI}, while this is not the case in $procSTWSN^{noT}$; hence, process $procSTWSN^{noT}$ cannot simulate $procSTWSN$. □

Proposition 7. The STWSN protocol is secure to attempts to fake the number of sessions.

As described in Section 4.4, in STWSN we allow packets to stay in the buffer until the end of the current session even if a new session (with the same source, destination, and application) is opened, which eliminates the faking session attack.

Proof. Let the constant BUFFEREMPTY represent the signal that an intermediate node I will send out on channel c_{signal} once its buffer has been emptied. Let $STWSN^{del}$ be the $STWSN$ protocol that deletes the buffer because of new session packets. Then, we prove that the bisimilarity

$$(procSTWSN^{del}, v^{init}) \approx_{pt} (procSTWSN, v^{init}).$$

does not hold, because in $procSTWSN^{del}$ there exists a a labelled trace that ends with the labelled transition $\xrightarrow{\overline{c_{signal}}\langle BUFFEREMPTY\rangle, d}$ after a packet of a new session has been received, while this is not the case in $procSTWSN$; hence, process $procSTWSN$ cannot simulate $procSTWSN^{del}$. □

6 | PERFORMANCE ANALYSIS

The ONE simulator [27] is used to compare the three protocols (*DTSN, STDP, STWSN*). All simulations used the same network scenario of chain of nodes between the source and destination. Each protocol was tested using the same parameters and run 10 times with different random number generator (RNG) seeds to negate systematic simulation affects. Table 1 lists the simulation settings. The scenario assumed a network with 5 nodes in a chain (a source, 3 intermediate nodes and a destination).

TABLE 1 Simulation parameters

Parameter Description	Value
World size	4500X3400
Node count	5
Simulation Update Interval	0.1 seconds
Network packet rate	10M per second
Simulation time	400,000 seconds
Transmit range	1500 meters
Buffer sizes tested	5MB
Message size	100KB
Protocols tested	*DTSN*, *STDP*, *STWSN*

The message time-to-live (TTL) was explicitly set to be greater than the total simulation time so that TTL did not affect the message delivery rate. Messages were only dropped due to queue overflow or protocol-based metrics. Finally, for the first three cases (Figures 7-9) we assumed a benign environment, without attacker, and only evaluated the overhead in normal operations.

Figure 7 compares of the three protocols in terms of the total data size (in bytes) for a whole session, as a function of the number of messages (data and control messages as well as header overhead) in that session. As expected, the message overhead of STWSN was the largest due to the hash-chain and the Merkle-tree and the size of the ACK/NACK messages were larger than for STDP. However, by increasing the number of messages, the total overhead for the session did not increase dramatically.

In Figure 8, we assumed a session with 1000 messages to examine the influence of the PRR (7.5%-12%). The figure indicates that in all cases the number of bytes we need to send increases with increasing PRR. Note that DTSN changed very slightly compared to SDTP and STWSN, because the volume of DTSN was much much smaller than the other two. Further, control messages in DTSN required small additional bytes since the ACK/NACK messages did not contain cryptographic add-ins. The ratio between STWSN and DTSN started 11 (PRR=7.5%) and increased to 13 (PRR=12%) while the ratio between SDTP and DTSN started at 6 (PRR=7.5%) and increased to 7 (PRR=12%). The ratio between STWSN and SDTP started at 1.9 (PRR=7.5%) and increased to 2 (PRR=12%). Thus, although STWSN requires more bytes for the session, the difference is not exponential. Furthermore, with more powerful sensors in the future, this overhead would still be within an acceptable level.

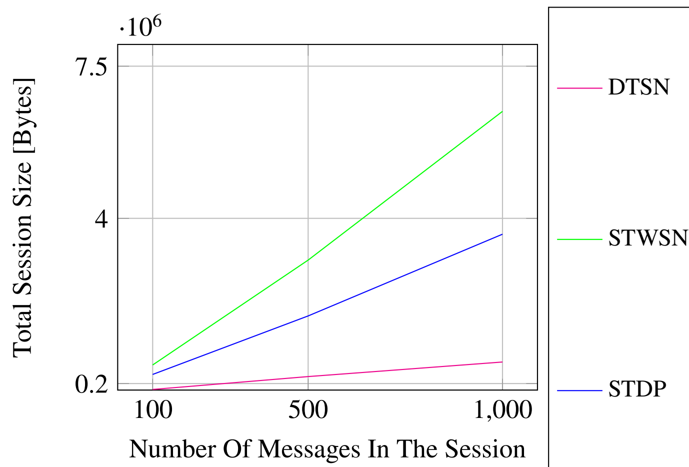

FIGURE 7 Comparison of the total data size (in bytes). We assumed a benign environment, without an attacker.

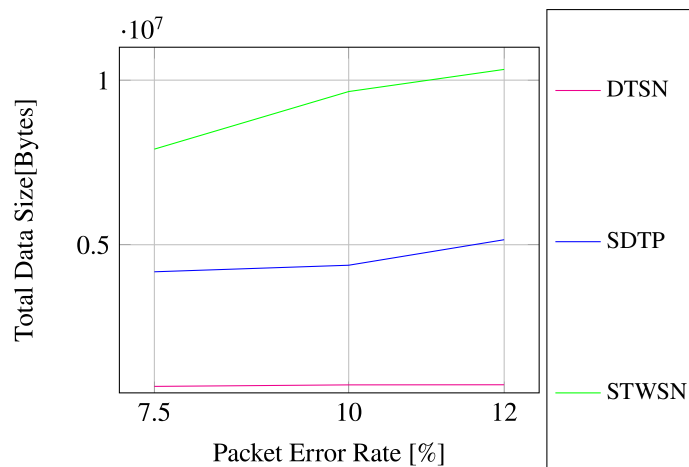

FIGURE 8 The number of bytes required to send 1000 messages as a function of PRR, in a benign environment.

Figure 9 focuses on the aggregation methods of STWSN that aim to mitigate the effect of forging an $NACK$ packet. Here, instead of forwarding individually the received ACK or $NACK$ messages, intermediate nodes wait for a short time, then aggregate the ACK or $NACK$ messages they receive within this period (Section 4.4). The figure shows that the influence on the average size of ACK messages was slight but was significant on the $NACK$ messages.

Next we simulated EAR attacks in a hostile environment. We assumed a network consisting of a chain of a source, 3 intermediate nodes, an attacker node and a destination node respectively in this order. Further, we assumed that there was no PRR rate (PRR=0), and set the number of messages in the session to 1000. The sender windows size was set to 15 messages (i.e. the sender sends a EAR message after each 15 messages).

We simulated three kinds of EAR attacks, namely:

- Attacker 1: Setting EAR bits. For each message, the attacker decides to set the originally unset EAR bits (piggybacked between the source and the destination) with probability P.
- Attacker 2: Unsetting EAR bits/Dropping EAR messages. For each message with EAR bit originally set or each EAR message, the attacker unsets the EAR bit or drops the EAR message with probability P.
- Attacker 3: Combined set and unset/drop attack. The attacker can perform any combination of the two attacks above.

We assumed that the attacker could performs each attack with a probability P (P=25%-75%).

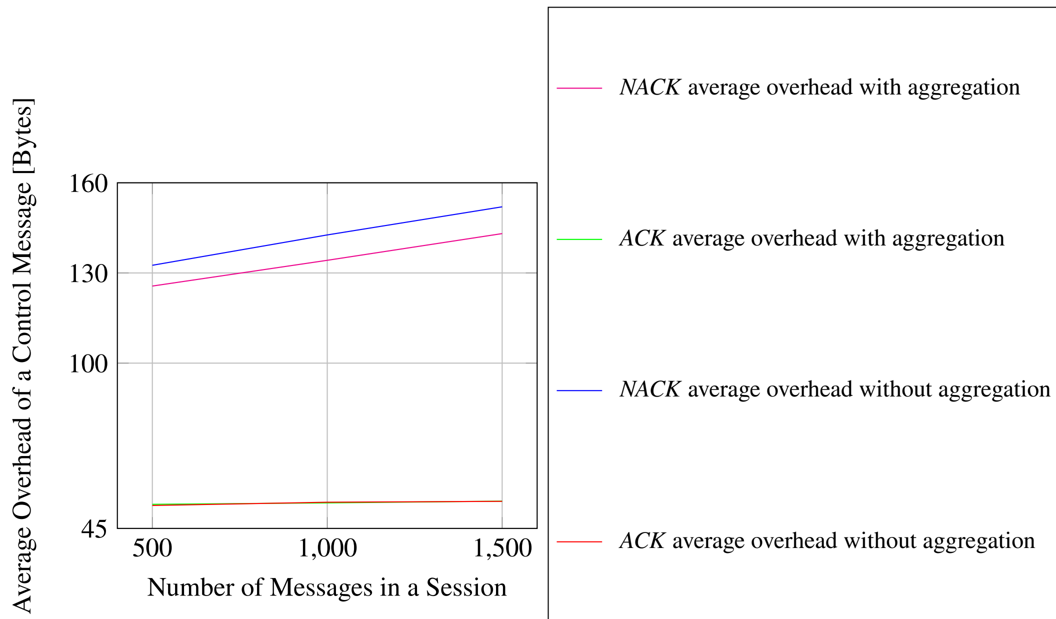

FIGURE 9 The average number of bytes required for an intermediate node to send a control message with and without aggregation methods. We assumed a benign environment, without an attacker and no PRR.

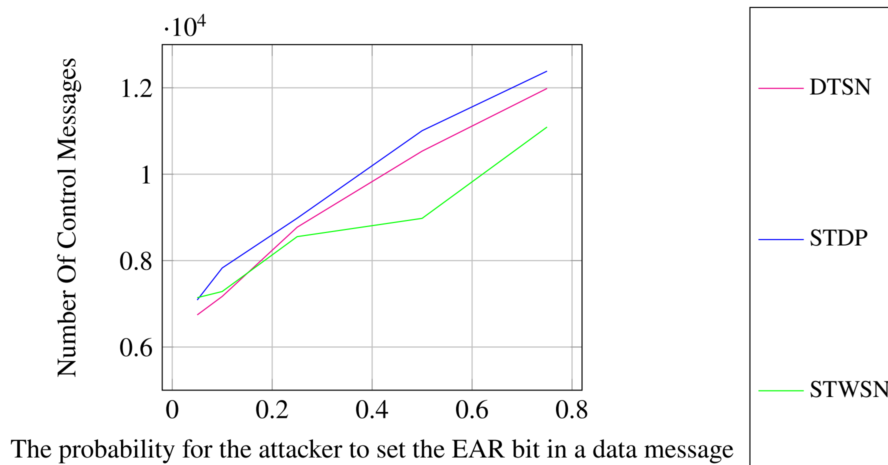

FIGURE 10 Comparison of the number of control messages, first attacker

Figures 10 - 12 depict the results of the three attacks, respectively. Figure 10 presents the influence of the first attacker, as the probability increased, our algorithm (STWSN) managed to have the lowest number of control messages (ACK, $NACK$) as a result of the control message aggregation method.

Figure 11, which addresses the second attacker, shows that as the attack probability increased the STWSN alone could continue functioning and deliver the messages, while the other two protocols closed the session because the unset EAR bits or dropped EAR messages prevented the destination from sending control messages.

In Figure 12, we simulated the second attacker again. The session time in this scenario was set to the previous session time of STWSN. In this setting, we examined how many control messages DTSN and STDP could still be send until they closed the session. The figure shows that the session closed after only a small number of control messages under DTSN or STDP whereas STWSN mitigated the attack and continued sending messages.

Finally, we simulated the third attacker that can unset or set (with coin tossing probability) EAR bits and/or drop EAR messages. Figure 13 depicts the result for the combined attack, and shows that our algorithm had fewer control messages as

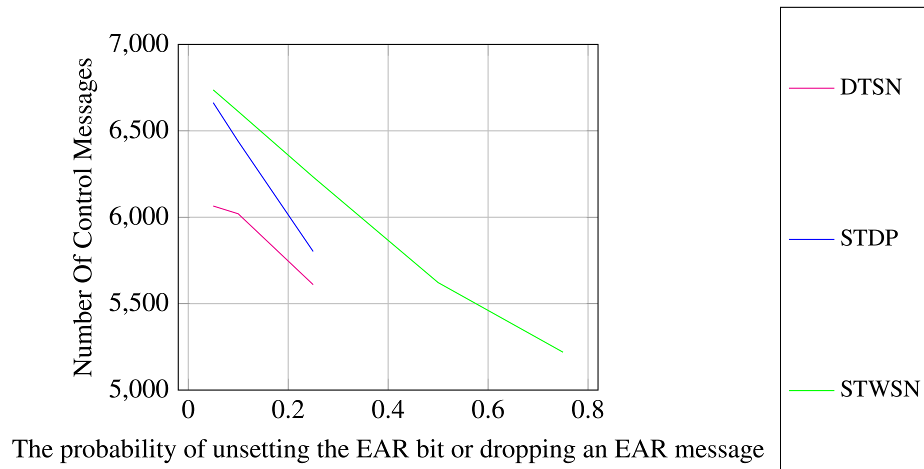

FIGURE 11 Comparison of the number of control messages, second attacker

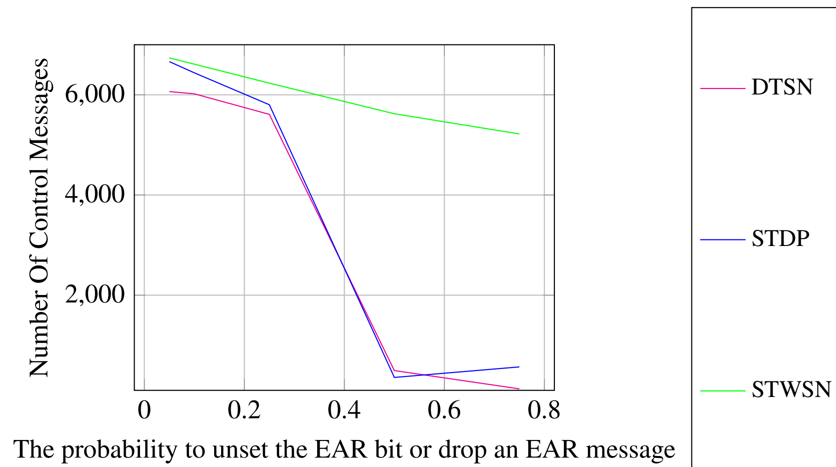

FIGURE 12 Comparison of the number of control messages, second attacker

the attack probability increased. Clearly, after P= 25% STWSN became the best algorithm; in addition the difference between P= 5% and P= 75% in this case was only 650 control messages, compared to SDTP and DTSN where the difference exceeded 1500 messages. It is also worth noting that our algorithm had the lowest simulation time, in other words, it was the fastest of the three in this setting.

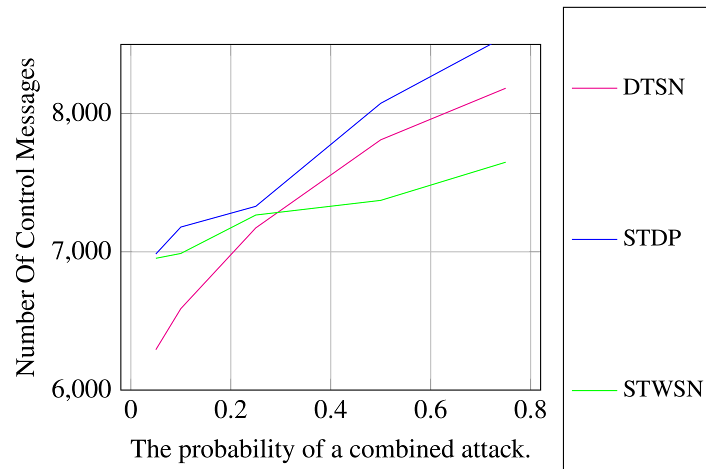

FIGURE 13 Comparison of the number of control messages, third attacker

7 | CONCLUSION

In this paper, we proposed STWSN, a new secure transport protocol for wireless sensor networks, which provides new security extensions to DTSN. The security mechanism of the new protocol is based on an efficient application of hash chains and Merkle-trees. We showed that given the proposed security mechanisms, STWSN resists reliability and energy efficiency requirement attacks, including SDTP attacks. We confirm the accuracy of our security analysis results by a formal method. Our planned future work addresses the implementation and validation of the proposed scheme in a WSN testbed environment.

ACKNOWLEDGMENTS

Amit Dvir was supported by a Marie Curie Mobility Grant, OTKA-HUMAN-MB08-B 81654. The work reported in the paper was developed in the framework of the project "Talent Care and Cultivation in the Scientific Workshops of BME" project.

References

[1] 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), pages 1–6, Mannheim, Germany, March 2010.

[2] L. Buttyan 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.

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

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

[5] D. Coppersmith and M. Jakobsson. Almost optimal hash sequence traversal. In Fourth Conference on Financial Cryptography, pages 102–119, Southampton, Bermuda, March 2002.

[6] R. C. Merkle. Protocols for Public Key Cryptosystems. In Symposium on Security and Privacy, pages 122–134, California, USA, April 1980.

[7] A. Dvir, L. Buttyan, and TV Thong. SDTP+:Securing a Distributed Transport Protocol for WSNs using Merkle Trees and Hash Chains. In IEEE International Conference on Communications, pages 1–6, Budapest, Hungary, June 2013.

[8] TV Thong and A. Dvir. On formal and automatic security verification of wsn transport protocols. Cryptology ePrint Archive, Report 2013/014, 2013.

[9] R. Ye, A. Boukerche, H. Wang, X. Zhou, and B. Yan. Resident: a reliable residue number system-based data transmission mechanism for wireless sensor networks. Wireless Networks, Aug 2016.

[10] M. Kordlar, G. Ekbatanifard, A. Jahangiry, and R. Ahmadi. A transmission method to guarantee qos parameters in wireless sensor network. In Leonard Barolli, Tomoya Enokido, and Makoto Takizawa, editors, Advances in Network-Based Information Systems, pages 801–811, 2018.

[11] M. Adeel Mahmood, W. K.G. Seah, and I. Welch. Reliability in wireless sensor networks: A survey and challenges ahead. Computer Networks, 79:166 – 187, 2015.

[12] A. Ghaffari. Congestion control mechanisms in wireless sensor networks: A survey. Journal of Network and Computer Applications, 52:101 – 115, 2015.

[13] Y. C. Hu, A. Perrig, and D. Johnson. Ariadne: A secure on-demand routing protocol for ad hoc networks. Wireless Networks, 11(1-2):21–38, Jan. 2005.

[14] A. Perrig, R. Canetti, D. Song, and D. Tygar. The TESLA broadcast authentication protocol. RSA Cryptobytes, 5(2):2–13, 2002.

[15] Y. C. Hu, D. B. Johnson, and A. Perrig. Secure efficient distance vector routing in mobile wireless ad hoc networks. Ad Hoc Networks, 1(1):175–192, July 2003.

[16] L. Lamport. Password authentication with insecure communication. Journal of the ACM, 24(11):770–772, Nov. 1981.

[17] N. Haller. The S/KEY one-time password system. RFC 1760, Internet Engineering Task Force, February 1995.

[18] Y. Matias and E. Porat. Efficient pebbling for list traversal synopses. In Thirtieth International Colloquium on Automata, Languages and Programming, pages 918–928, Eindhoven, Netherlands, July 2003.

[19] A. M. Ben-Amram and H. Petersen. Backing up in singly linked lists. In STOC, pages 780–786, Georgia, USA, May 1999.

[20] R. Merkle. Secrecy, authentication and public key systems / a certified digital signature. Ph.D. dissertation, Dept. of Electrical Engineering, Stanford University, 1979.

[21] L. Kissner and B. Laurie. Google wave protocol - general verifiable federation. Google white paper, May 2009.

[22] A. Herzberg and H. Shulman. Stealth DoS Attacks on Secure Channels. In Network and Distributed System Security Symposium, pages 1–19, California, USA, Feb 2010.

[23] S. Turner and L. Chen. Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms. RFC 6151, , Internet Engineering Task Force, March 2011.

[24] L. Buttyan and JP Hubaux. Security and Cooperation in Wireless Networks: Thwarting Malicious and Selfish Behavior in the Age of Ubiquitous Computing. Cambridge University Press, New York, NY, USA, 2007.

[25] R. Roman, C. Alcaraz, and J. Lopez. A Survey of Cryptographic Primitives and Implementations for Hardware-Constrained Sensor Network Nodes. Mobile Networks and Applications, 12(4):231–244, Oct. 2007.

[26] A. Menezes, P. C. van Oorschot, and S. Vanstone. Handbook of Applied Cryptography. CRC Press, 1997.

[27] Ari Keränen, Jörg Ott, and Teemu Kärkkäinen. The ONE Simulator for DTN Protocol Evaluation. In STT, March 2009.

[28] P. Ganesan, R. Venugopalan, P. Peddabachagari, A. Dean, F. Mueller, and M. Sichitiu. Analyzing and modeling encryption overhead for sensor network nodes. In Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications, pages 151–159, CA, USA, Sep. 2003.

[29] K. Piotrowski, P. Langendoerfer, and S. Peter. How public key cryptography influences wireless sensor node lifetime. In Proceedings of the fourth ACM workshop on Security of ad hoc and sensor networks, pages 169–176, VA, USA, Nov. 2006.

8 | APPENDIX A - DTSN

In DTSN [4], within a session, each data packet is given a sequence number n. DTSN provides reliable packet delivery by using the three control packets: (1) (EARs) packet - Explicit Acknowledgment Requests, (2) (ACKs) packet - Acknowledgments , (3) ($NACK$s) packet - Negative Acknowledgments. The EAR packets are sent by the source node, whereas ACK and $NACK$ messages are sent by the destination node. Each EAR packet can be sent in the following cases (by the source node either as an independent packet or in the form of a piggybacked data packet): (i) when the source has already transmitted a predefine number (window) of data packets; (ii) the buffer of the source becomes full; (iii) the source does not receive any data transmission requests from the upper layer application after a timeout period which predefined; or (iv) upon expiration of the EAR timer (EAR_timer) [4]. Note that, we assume that EAR is always in the form of a piggybacked bit flag, to decrease traffic overhead [4]. Moreover, intermediate nodes cache data packets with some probability p, and they also can retransmit both data packets and control messages.

The destination, sends control packets (ACKs or $NACK$s) according to the received data packets. Using those control packets, the destination inform that intermediate nodes and the source node that packets arrived, e.g. by sending an ACK packet with sequence number n indicates that packets with sequence numbers smaller than or equal to n have been received by the destination node. The $NACK$ packet has two parts; the first is a sequence number and the second is a bitmap of set/unset bits where the sequence number n indicates the acknowledged information as in the case of ACK, and each set bit in the bitmap corresponds to a data packet that did not reach the destination, and needs to be retransmitted.

As explained above, for retransmission purposes, intermediate nodes store data packets with some probability p. Upon receiving an ACK packet with sequence number n, an intermediate node will delete all the stored data packets with sequence numbers less than n, and then will forward the ACK packet towards the source. Upon receiving a $NACK$ packet, an intermediate node handles the included sequence number in the same way as for the ACK packet, and in addition, it retransmits each data packet based on the corresponding set bit in the bitmap. For each retransmitted data packet, the intermediate node unsets the corresponding bit in the bitmap, and then forwards the updated $NACK$ message towards the source. In the case where all the bits in the bitmap are unset the $NACK$ becomes an ACK [4].

9 | APPENDIX B - STWSN NODES' MECHANISMS

9.1 | Source Node Mechanism

Algorithm 1 depicts the source node algorithm.

Algorithm 1 Source Mechanism

```
1: Open Session Counter= 0
2: Max Open Tries = N                                           ▷ N can be 10
3: session open = false
4: MaxSN = 0
5: function SOURCE
6:     if Open Session () then      ▷ Send and Received are triggered by the open session
7:         while The entire data acknowledged by the receiver do
8:             Send () or Received ()
9:         end while
10:    end if
11: end function
12: function OPEN SESSION
13:    while !(session open) do
14:        Compute session master secret K, KACK, KNACK
15:        Generate hash chain                                  ▷ ACK authentication values
16:        Compute ACK authentication values: K(i)_ACK = h(K(i + 1)_ACK)
17:        Generate Merkle-tree                                 ▷ NACK authentication values
18:        Compute NACK authentication values: K(n)_NACK = PRF(KNACK, perpacketNACKsecretε, n)
19:        Sign the OpenSession-Packet
20:        Send OpenSession-Packet                              ▷ see Fig 4
21:        OpenSession-Counter++
22:        if OpenSession-Counter == Max Open Limit then return false
23:        else
24:            Launch OpenSession-Timer and Wait
25:            if Ack Received for the OpenSession-Packet then
26:                if ACK verification OK then return true
27:                end if
28:            end if
29:        end if
30:    end while
31: end function
32: function SEND
33:    while Sending Window is not full do
34:        Send Packet (packet index, data)
35:        SendingWindow-Counter++)
36:    end while
37: end function
38: function RECEIVED
39:    if ACK_i message then                                    ▷ Compute hash for ACK_i
40:        Hash_comp = hash the ACK authentication value i times
41:        if Hash_comp==root then
42:            Acknowledge the packets with seq numbers <= i
43:            MaxSN = i
44:        else
45:            Drop ACK message
46:        end if
47:    else if NACK message then        ▷ Check the ACK part from the NACK msg, see Fig 14
48:        Hash_comp = hash the ACK authentication value i times
49:        if Hash_comp==root then
50:            Acknowledge the packets with seq numbers <= i
51:            MaxSN = i
52:        else
53:            Drop NACK message
54:        end if
55:        for each set bit in bitmap do
56:            if NACK authentication values verification using the Merkle-tree then
57:                Retransmits the required/corresponding packets
58:            end if
59:        end for
60:    end if
61: end function
```

9.2 | Destination Node Mechanism

Algorithm 2 depicts the destination node algorithm.

Algorithm 2 Destination Mechanism

```
1: function RECVOPENSESSIONPACKET                       ▷ Upon receiving a OpenSession-Packet
2:     if signature OK then
3:         Generate session parameters   ▷ the ACK master secret, the NACK master secret, the session master secret, the Merkle-tree, and the hash chain
4:         Sends an ACK
5:     end if
6: end function
7: function RECVDATAPACKET(Data_i)                      ▷ Upon Receiving a Data Packet Data_i
8:     if Authentication OK then
9:         Deliver the packet to the upper layer
10:        if EAR then
11:            if destination EAR timer not finished then
12:                if X < MaxControlMessages then
13:                    if All messages in the window received then
14:                        Send ACK                       ▷ extended with the ACK authentication value
15:                        X++
16:                    else
17:                        Send Nack   ▷ extended with the ACK authentication value and for every missing message j-th bit is set in the bitmap and siblings added
18:                        X++
19:                    end if
20:                end if
21:            end if
22:        end if
23:    end if
24: end function
25: function EARTIMEREXP                                 ▷ Upon EAR_timer Expired
26:    Set X = 0
27: end function
28: function STATUSTIMEREXP                              ▷ Upon Status_timer Expired
29:    if All messages in the window received then
30:        Send ACK                                       ▷ extended with the ACK authentication value
31:    else
32:        Send Nack   ▷ extended with the ACK authentication value and for every missing message j-th bit is set the bitmap and siblings added
33:    end if
34: end function
```

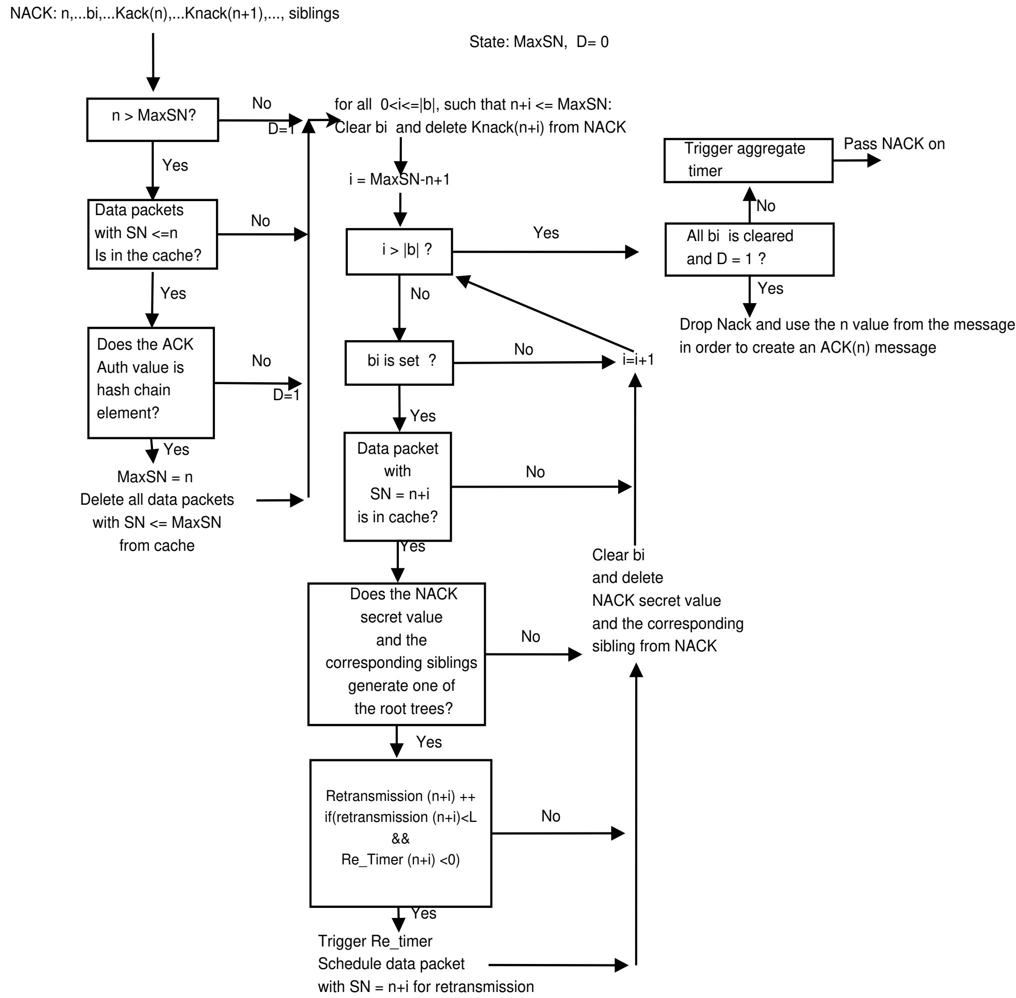

FIGURE 14 The intermediate algorithm, focusing on the $ACK/NACK$ processing.

9.3 | Intermediate Node Mechanism

For detailed processing of the intermediate node algorithm see Fig. 14.

10 | APPENDIX C - OVERHEAD ANALYSIS

The evaluation of the overhead of our new security scheme based on the building blocks, one Merkle tree compared to many, as well as retransmissions.

10.1 | Building Block Overhead

We evaluate the overhead of our new security scheme in terms of the building blocks alone. Our evaluation assumes MICAz motes and is based on a thorough evaluation of works and benchmarks on the overhead of cryptographic operations applied in MICAz motes [25, 28]. Finally, we calculate the time overhead of the security scheme for each node based solely the building blocks (without the overhead of the timers).

For simplicity, we present the overhead computation of the hash chain and Merkle-tree separately which are the building blocks of our security scheme. To publish the root in a secure way we can use a digital signature; e.g., RSA [29] or Elliptic

TABLE 2 Estimated Time Equations for the Security Scheme (building blocks only).

	Source	Intermediate
Hash	$ECC_G + (m+1) \cdot SHA$	$ECC_V + \frac{(m+1) \cdot m}{2} \cdot SHA$
Merkle-tree	$ECC_G + (2^D - 1 + m) \cdot SHA$	$d \cdot m \cdot loss_propability \cdot SHA$

Curve DSA [25], and to authenticate the packet we can use HMAC [23]. Therefore, one HMAC operation [23] is equivalent, in the worst case, to 4 hash operations(two main hash operations where each has to hash two blocks).

The source node and destination node need to generate a hash chain with length of $m+1$ [28]. However, only the source node has to sign the first packet by using Elliptic Curve DSA [25]. As explain above, each intermediate node and destination have to verify the signature once and each intermediate node has to verify the hash element per each ACK packet. Thereofre, one signing operation ECC_G and $(m+1) \cdot SHA$ time is required at the source node ($ECC_G + (m+1) \cdot SHA$), and one verification operation ECC_V and $(m+1)$ hashing operations at the destination node; i.e., $(m+1) \cdot SHA$ time. At intermediate nodes, one verification operation ECC_V and in the worst case, where the probability of storing a data packet is 1 and the size of AW windows at the source is set to 1, $\frac{(m+1) \cdot m}{2}$ hashing operations are required. Thus, the worst case time overhead at the intermediate node is $ECC_V + \frac{(m+1) \cdot m}{2} \cdot SHA$.

The source node generates a binary Merkle-tree of height D (for simplicity, we assume the case where $t = 1$). Furthermore, to create the leaves, the source node and destination node require $m \cdot SHA$ time [28], whereas in order to send the tree roots in a secure manner requires two signing operations ECC_G and ECC_V. To generate the binary Merkle-tree, the source node hashes at each level of the binary tree, which takes $2^D - 1$ hash operations that take $(2^D - 1 + m) \cdot SHA$ time. For each bit in the bitmap of a $NACK$ control packet, the intermediate node that stored the data packet needs to verify the $NACK$ authentication value, which requires d hash operations. With a given loss probability, the intermediate nodes on the path between the source node and the destination node have to retransmit $d \cdot m \cdot loss_probability$ times, which requires $d \cdot m \cdot loss_probability \cdot SHA$ time (in the worst case where an intermediate node stores every packet, the storage probability is equal to 1). Table 2 summarizes the time overhead estimation of our new scheme.

Regarding packet overhead, for the ACKs, it is the hash value size (for each ACK packet we need to add one hash value (64/128/256 bits)). Therefore, in the worst case the packet overhead is $m \cdot hs$, where hs is the size of the hash value (number of bits of the hash field); for the $NACK$s, it is the number of set bits in the bitmap times the authentication value size (which is d times the hash value size). Hence, an intermediate node needs to handle $d \cdot loss_probability \cdot SHA$ packet overhead.

10.2 | Several Merkle-trees versus one Merkle-tree

One of the issues regarding Merkle-trees is the communication overhead (number of siblings that need to be sent to reveal a key). In the following, we show that using several smaller Merkle-trees with an optimized height is more effective than using only one large Merkle-tree. The reason why we use several, t, smaller Merkle-trees of height d instead of one large Merkle-tree of height D for authentication is that in most cases the latter imposes a larger overhead. Note that the number of leaves in the large Merkle-tree and the t smaller trees is m, $m = t \cdot 2^d = 2^D$. As Fig. 15 illustrates, in the case of using one Merkle-tree of height D, the number of leaves (m) is 2^D, whereas in the case of small Merkle-trees of height d $(D > d)$, each small tree has 2^d leaves. Hence, it is easy to see that the number of small trees is $t = 2^{D-d}$. Figure 16 shows the influence of the number of Merkle-trees over the communication overhead in a scenario of a chain network with 800 messages from the source to the destination with a packet error rate of 5%. The figure indicates that the $NACK$ overhead decreases when the number of trees increases.

Let us assume that a communication between source and destination needs to be authenticated via Merkle-trees. The total communication overhead during the authentication procedure is the number of small trees (2^{D-d}) multiplied by the number of sibling nodes (d) per leaf and the probability for a data packet to be lost in the path between the source and the destination($\frac{1}{1-\alpha}$). Therefore, the total communication overhead is $C_c \cdot \alpha \cdot d \cdot 2^d \cdot 2^{D-d}$, where C_c is the communication unit cost (i.e., the cost of each communication). In addition, the storage overhead due to the fact that at the beginning the intermediate nodes need to store

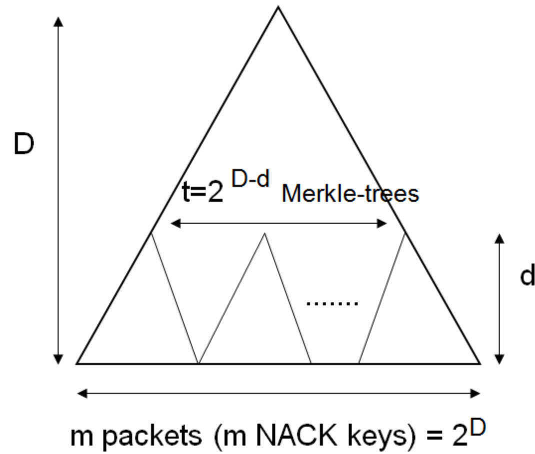

FIGURE 15 The difference between using one large Merkle-tree of height D and several smaller Merkle-trees of height d.

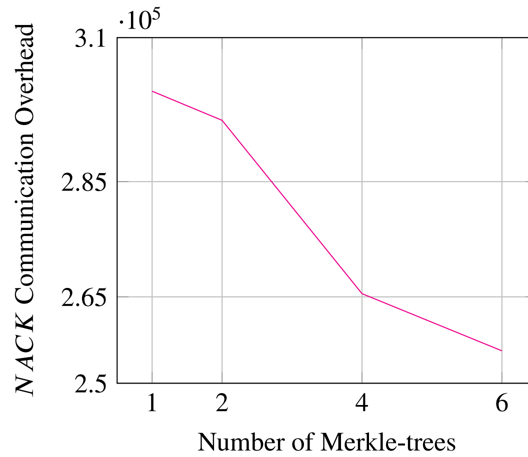

FIGURE 16 An example of the influence of small Merkle-trees compared to one large tree, for a chain network with 800 messages from the source to the destination with a packet error rate of 5%

the root hash value of each Merkle-tree is $C_s \cdot 2^{D-d}$, where C_s is the storage unit cost. Hence, the total overhead using several Merkle-trees ($F(d)_{smalltrees}$) can be defined as a function of d:

$$F(d)_{smalltrees} = C_c \cdot \frac{1}{1-\alpha} \cdot d \cdot 2^d \cdot 2^{D-d} + C_s \cdot 2^{D-d}. \tag{2}$$

We denote the total overhead as a function of d for the purpose of computing the optimal value of d.

When using one large Merkle-tree, we have minimal storage overhead because we only need to store the root hash value. The communication overhead is imposed by revealing the authentication value (secret values and their siblings). In particular, the total overhead ($F(d)_{BigTree}$) using one Merkle-tree is presented in Eq. 3.

$$F(d)_{BigTree} = \frac{1}{1-\alpha} \cdot C_c \cdot D \cdot 2^D + C_s. \tag{3}$$

From these, we can calculate that the overhead of several trees is less than the case of a large tree when:

$$
\begin{aligned}
C_c \cdot \frac{1}{1-\alpha} \cdot d \cdot 2^d \cdot 2^{D-d} + C_s \cdot 2^{D-d} &< \\
\frac{1}{1-\alpha} \cdot C_c \cdot D \cdot 2^D + C_s &=> \\
C_c \cdot \frac{1}{1-\alpha} \cdot d \cdot 2^D - \frac{1}{1-\alpha} \cdot C_c \cdot D \cdot 2^D &< \\
C_s - C_s \cdot 2^{D-d} &=> \\
\alpha < 1 - \frac{C_c \cdot 2^D(D-d)}{C_s(2^{D-d}-1)} &
\end{aligned}
\tag{4}
$$

Intuitively, when the loss probability is less than the derived threshold, it is better to use several smaller trees. In addition, the threshold may be small in realistic cases; hence, it is always better to use several smaller trees. Now that we have clarified why we apply small Merkle-trees instead of one large Merkle-tree, we can present the computation of the optimal value of d. The optimal d has to fulfill the equation $F'_{smalltrees}(d) = 0$, where $F'_{smalltrees}(d)$ is the derivation of $F_{smalltrees}(d)$. More precisely, from

$$F_{smalltrees}(d) = C_c \cdot \frac{1}{1-\alpha} \cdot d \cdot m + C_s \cdot 2^{-d} \cdot m$$

we have

$$F'_{smalltrees}(d) = C_c \cdot \frac{1}{1-\alpha} \cdot m - C_s \cdot ln2 \cdot 2^{-d} \cdot m$$

Then the optimal d is

$$d^{opt} = log_2\left(\frac{C_s \cdot ln2 \cdot \frac{1}{1-\alpha}}{C_c}\right), d < D. \tag{5}$$

Note that Eq. 5 cannot be independent, since otherwise it would be a case where the optimal d is larger than D. Finally, note that the values of C_c and C_s depend on the specific implementation, and the value of α can be set by the source based on the network quality it detects during a session. Therefore, based on parameters (C_c, C_s, α) and Eq. 5, the source can decide to build one or several Merkle-trees.

10.3 | Futile Retransmission Overhead

Thus, we only need to reason about the overhead of futile retransmissions in the worst case. Let us define p_a as the probability an attacker will capture a data packet, p, as the probability an intermediate node will store a data packet, N_{path} as the average number of nodes in the path between a source and a destination, L as the maximum number of retransmissions per packet, and m as the average number of packets in a session. These definitions can be found in Table 3. In order to calculate the overhead, we need to take the average number of intermediate nodes that will be stored any given packet into account:

$$p \cdot N_{path}. \tag{6}$$

The worst case of transmitted packets:

$$p_a \cdot L \cdot m. \tag{7}$$

Therefore, the overhead upper bound is:

$$p_a \cdot p \cdot N_{path} \cdot L \cdot m. \tag{8}$$

Essentially, the network does not have any information on the attacker. Hence, the network has no control over parameters p_a and N_{path}, but only over parameters p, L, and m. Thus, the tradeoff between these three parameters should be considered.

□

p_a	The probability an attacker will capture a data packet
p	The probability an intermediate node will store a data packet
N_{path}	The average number of nodes in the path between a source and a destination
L	The maximum number of retransmissions per packet
m	The average number of packets in a session

TABLE 3 List of abbreviation for retransmission overhead in the case of a $NACK$ replay attack