SecSyWiSe: A Secure Time Synchronization Scheme in Wireless Sensor Networks (Draft)

Johannes Barnickel and Ulrike Meyer
UMIC Research Center
RWTH Aachen
Email: http://itsec.rwth-aachen.de/people

September 12, 2009

Abstract

In this paper we suggest a protocol for secure data distribution in a one-way communication scenario aimed at time distribution for sensor networks. Our protocol is designed to support timestamps in any format, including repeating, resettable, and incomplete timestamps. The protocol requires a trusted powerful source node. We show that the protocol is immune to attacks including replay attacks, wormhole attacks, byzantine traitors, and node capture. These security features are achieved by the use of digital signatures with additional replay protection that is not based on time synchronization. Pulse delay attacks can be countered by the use of a filter accompaigned by a continuous timestamp format. The protocol relies on the source node to sign and broadcast messages with sufficient transmission power to reach all other nodes. These will not transmit any messages for time synchronization. We show that our protocol scales to networks of any density and allows for stealth applications. Conservative simulation results show that our protocol is more energy efficient for the sensor nodes than a comparable protocol.

1 Introduction

Time synchronization in sensor networks is important for many applications. For power saving, most stations are in a power saving sleep mode most of the time. They can only communicate with each other when their radio interfaces are switched on. Radio communication is the single most power consuming task of sensor nodes. A sensor node that cannot find out when to try to communicate to other stations will either drop from the network or need to try over and over again, which greatly reduces its battery life. With preciser time synchronization, the wake up schedule will be executed more precisely. In addition, in many applications, data gathered by sensor nodes is meaningless if it cannot be bound to the time when it was gathered. For example, precise time tagging and thus synchronization is required for beam forming, tracking, and locating an object.

Time synchronization must resist modification attempts by attackers, especially in applications such as accounting, metering, and billing systems. In these applications, an attacker could gain direct profit beyond obstructing services from tampering with the time synchronization process.

In this paper we suggest a novel time synchronization protocol that is secure against various attacks. As opposed to previously suggested security aware protocols for time synchronization, our protocol does not reveal the network structure to an observer. Also, our protocol does not require nodes to have established keys with their neighbors. A single static key must be known to all stations on the network, yet our protocol is immune to node capture. The protocol also allows for the use of incomplete or repeating timestamps while preserving most security properties. If stations are added to an existing network, they can use the periodic time synchronization message to synchronize to the network such that no additional messages are required for any nodes.

We compare our protocol to TinySeRSync, a state-of-the-art time synchronization protocol for sensor networks, using a simulation of energy use and node workload. The results show that our protocol, depending on the signature scheme and parameters used, offers similar or better energy efficiency and causes comparable workload on each node.

Our protocol was initially designed for secure time synchronization in sensor networks. However, it could also be used to secure other unprotected radio transmitted time signals such as WWV, DCF77, HBG[1] etc. Another possible application of the proposed protocol is public key distribution in broadcast scenarios, such as pay television.

The rest of the paper is structured as follows: In Section 2, we provide an overview on related work. In Section 3, we explain how our protocol works. In Section 4, we discuss the design choices and the security features of our protocol. In Section 5, we evaluate the performance of our protocol. In Section 6, we discuss various deployment options and potential optimizations of the protocol. The conclusion is drawn in Section 7. Finally, in Section 8, we provide an outlook on future work.

2 Related Work

Synchronization protocols aimed at wired networks and large computers cannot be used for sensor networks because they lack permanent Internet connectivity and have to meet high constraints regarding energy efficiency and computational power. For example, the standard time synchronization protocol for Internet-enabled devices NTP [14] is not feasible for sensor networks because it requires certificate chains and reliable two-way communication. As a consequence, designing time synchronization protocols for sensor networks is an active research

[1]US, German, and Swiss legal time radio sending stations

topic. Most of the protocols suggested so far were not designed with security in mind. Only more recent protocols take countering certain attacks into consideration.

The most well-known insecure protocols are the Reference Broadcast Synchronization [4], the Timing-Sync Protocol for Sensor Networks [5], and the Flooding Time-Synchronization Protocol [12]. All of them are vulnerable to node capture and impersonation [6, 11], i.e., a node that is controlled by an attacker is able to impact synchronization precision.

Protocols that take attacks into consideration are usually based on symmetric cryptography and explicitly avoid the use of asymmetric cryptography. These protocols use one of two approaches: Message authentication codes (MACs) with pairwise keys or variants based on the TESLA protocol.

If pairwise keys are used (e.g., [6, 8, 10]) the nodes are required to establish keys with each of their neighbors. Establishing these keys after deployment may lead to difficulties in hostile environments as attackers may influence key establishment. If the keys are already embedded on the nodes, the nodes must be deployed in a certain pattern so that they are neighbors of the nodes they share keys with. Requiring nodes to share keys with their neighbors also is an obstacle to node movement and to adding stations to an existing network. More complex key distribution schemes have been proposed, e.g., random key schemes in which any node shares predistributed keys with its neighbors with some probability [9]. While these schemes help to solve the problem of adding nodes to the network, the consequences of node capture are enhanced as a captured node will typically also store more keys than the ones shared with its neighbors. Node capture enables an attacker to act like a legit node, i.e., to influence the synchronization between neighboring nodes. Also, messages are flooded through the network and may be altered, delayed or dropped by captured nodes, thus influencing synchronization between more nodes than their direct neighbors.

Using message authentication codes with keys derived in a hash chain and delayed broadcast key disclosure (μTESLA) achieves some security goals typical for asymmetric signatures but requires time synchronization in the first place [15]. TinySeRSync [10] is a two-phase time synchronization protocol for TinyOS of this type. In the first phase neighboring nodes establish a rough common time by using pairwise synchronization protected with MACs. This requires preestablished pairwise keys. TinySeRSync makes no attempt at preventing node capture attacks in this Phase. Pulse delay attacks on the first phase are limited by the use of a maximum one-way transmission delay. However, there are no details given on how it has to be chosen and how large the influence of an attacker can remain. After the first phase (pairwise) was executed d times, the nodes have established a loose time synchronization. The second phase is called global synchronization. Broadcast messages are used for synchronization that are authenticated with μTESLA. Both phases are executed in intervals, with the pairwise phase being executed d times before each global synchronization. Pulse delay attacks on the second phase are claimed to be prevented by the synchronization established in the first phase. However, by attacking the first phase, e.g., with node capture, an attacker could introduce arbitrary data and other nodes would accept it as authentic data, especially new nodes. Moreover, there are claims that the pairwise synchronization in the first phase is not necessary if a filter is used and less than half of the nodes are captured [18]. However, the reasoning for these claims is lacking clarity.

As opposed to the protocols previously discussed, our protocol is not vulnerable to node capture as it does not use secret keys on the nodes and it does not require prior time synchronization. Also, it does not require pairwise keys between nodes.

Many protocols [6, 8, 10] use filters to avoid accepting time synchronization messages that are probably inconsistent and therefore faked or delayed. This is done by defining a maximum derivation from the local clock. A synchronization attempt that would change the local clock by a value greater than the preset threshold will be aborted. While it can be used to thwart any type of attack, this technique is often used to mitigate pulse delay attacks. We use filters in our protocol when continuous timestamps are used.

With respect to the underlying network model, our protocol is most similar to the insecure Flooding Time-Synchronization Protocol [12], which also uses continuous unidirectional broadcast synchronization with time offsets for all nodes on the network. Our protocol, however, additionally provides protection against attacks by the use of counters, digital signatures, filters, and by assuming a powerful source node rather than allowing nodes to forward synchronization messages. Additionally, our protocol supports non-continuous time stamps.

The feasibility of signature verification on sensor nodes was shown, e.g., by [16]. The 16 bit TelosB platform is capable of verifying a 1024 bit RSA signature in 220 ms and an ECDSA 160 one in 1.02 seconds. Considering lifetime, 2 AA batteries (2500 mAh) will last 2.5 million verifications with RSA 1024 bit and 543,916 with ECDSA 160 bit. We draw on the these results in both the protocol design itself and the evaluation of its performance.

3 New Protocol

3.1 Requirements and Scenario

The goal of the protocol is to achieve time synchronization on a wireless sensor network by periodically distributing timestamps to all sensor nodes in the network. The protocol shall be independent of the format used for the timestamp such that the format can be adapted to the individual needs of an application. In particular, the protocol shall support continuous timestamps (i.e., timestamps that always grow at the same rate) as well as non-continuous timestamps, such as timestamps without a date.

An attacker shall not be able to make a sensor node set its time forth or back with respect to the base time as broadcasted by the source node.

The source shall authenticate all time synchronization messages such that all nodes in the sensor network are able to verify the authenticity of these messages. The messages shall be protected against replay. A sensor node shall not accept

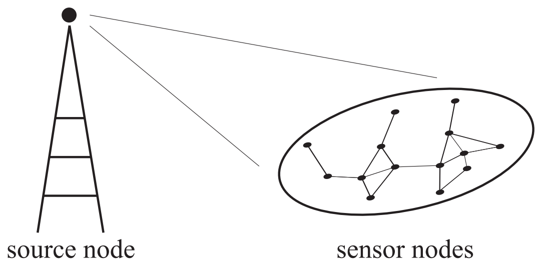

Figure 1: source node outside of sensor network

manufactured or replayed synchronization messages. The synchronization process shall recover quickly from a blocked channel, i.e., after a denial-of-service attack.

We assume a one-way communication scenario where a single source node, possibly in a remote location, is able to broadcast messages to all sensor nodes of a network. This scenario is illustrated in Figure 1.

The nodes do not require a communication channel back to the source node. The source node shall not need to know if a certain node has received the message or how many (if any) nodes have received the message at all. The source node uses considerably more energy than the other nodes because it is transmitting data periodically while the receiving nodes never send any time synchronization messages. Our protocol does not stop regular nodes from mutually exchanging data, except at the time and on the frequency used for time synchronization.

3.2 Protocol Description

The source node is in possession of a private/public signature key pair. All nodes in the sensor network are in possession of the corresponding signature verification key. The source node periodically broadcasts time synchronization messages to all clients. Each of these messages m contains an ID to identify the source node, a timestamp tsp, a counter value c, and a signature sig_s over the hash of tsp and c computed with the secret key s of the source node:

$$m = (ID, tsp, c, sig_s(h(tsp|c)))$$

The time stamp tsp is chosen such that it becomes current at the end of the transmission of the message.

Let Δ_{sig} be the time needed for generating the signature. Let Δ_{prep} be the delay for preparing the transmission and let δ_{wave} be the wave propagation delay between the source node and a sensor node. The synchronization messages are sent in fixed intervals.

In the following, we assume that the source node starts transmitting m at $T1$ and finishes at $T2$ (see Figure 2).

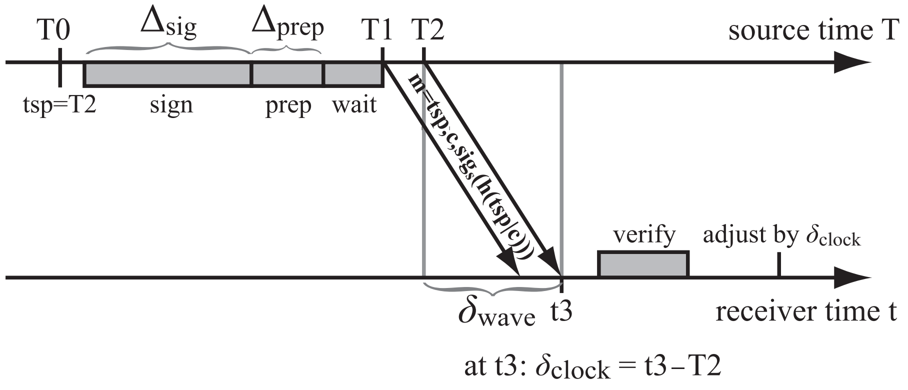

Figure 2: Timeline for synchronization to $T2$. The source node starts creating the new message m at $T0$, starts sending m at $T1$ and finishes sending m at $T2$. The receiver node has finished receiving m at $t3$.

The source node must start preparing the new synchronization message at $T0$ such that the packet will be ready for transmission by $T1$. $T0$ must be chosen such that:

$$T0 < T1 - \Delta_{sig} - \Delta_{prep}$$

At $T0$, the source node performs the following actions:

1. Increase the message counter c by one.
2. Set tsp to be the timestamp of $T2$.
3. Calculate $sig_s(h(tsp|c))$.
4. Assemble packet $m = (tsp, c, sig_s(h(tsp|c)))$.
5. Prepare the MAC layer for transmission.
6. Wait for $T1$.
7. Push message m to the channel with sufficient transmission power so that all stations on the network are able to receive it directly.

The counter value c is increased by the source node with every message sent and is used for replay protection.

The individual nodes intending to synchronize their clocks must already be listening when the synchronization message is pushed on the channel. All stations will receive the same signal at almost the same time, only limited by wave propagation differences.

After receiving the message, the individual nodes will start to verify the message m. The recipients will accept messages that contain a higher counter value than the last message they received, as well as a valid signature over timestamp and counter. Filters to prevent pulse delay attacks can be applied when the timestamp is continuous.

To achieve greater accuracy, the difference δ_{clock} between the client's own clock and the freshly received signal must be calculated immediately after receipt of the timestamp, i.e., before the signature is verified. This will ensure that the signature verification time does not add to the synchronization error. Thus, different sensor hardware platforms that need a different amount of time for the verification will synchronize to the same time. δ_{clock} will only be incorporated into the node's system time after signature and counter verification succeeded.

At $t3 = T2 + \delta_{wave}$, the message m has arrived at the receiver and is being processed as follows:

1. Extract the timestamp tsp from the message m and compute the clock difference $\delta_{clock} = t3 - tsp + \delta_{wave}$.

2. Extract ID from the message m and abort if it does not correspond to the trusted source node.

3. For applications that use a continuous time format, abort if δ_{clock} exceeds a threshold value (filter).

4. Extract the counter c from the message m, verify that the received counter c is greater than the local counter and abort if the verification fails.

5. Extract the signature $sig_s(h(tsp|c))$ from the message m, verify $sig_s(h(tsp|c))$ and abort if the verification fails.

6. Set the local counter to c.

7. Adjust the local time by δ_{clock}.

The synchronization error is going to be in the order of $\delta_{wave} = t3 - T2$. The wave propagation delay δ_{wave} between the source node and a sensor node is going to be around 1 μs per 300 meter according to the speed of light. This delay can always be predicted for a single node when if location is known. In some scenarios, it can be predicted to be similar for all nodes in the network, e.g., if the source node is located far away from a small network. In these cases, a signal for $T2 + \delta_{wave}$ could be transmitted at $T2$ to increase the precision. Although it does not affect the precision of the synchronization between the nodes, it increases the precision of the synchronization between the nodes and the source time.

4 Discussion of the new Protocol

4.1 Selection of Cryptographic Primitives

We want to avoid that captured nodes have an impact on other nodes. An attacker might be able to compromise at least one node and read or modify its stored data. Using symmetric message authentication codes with a pre-established key on the broadcast messages would allow an attacker to extract the symmetric key from a captured node and thus enable him to act just like the source node: Being able to transmit messages to all nodes with them accepting these messages, which would break all security goals of the system.

When using asymmetric signatures, an attacker that can read a receiving node's internal data cannot impersonate the source node. An attacker that is able to physically modify a receiving node cannot achieve more than to convince this single node that messages sent by himself are legitimate.

We chose not to use a delayed key disclosure symmetric broadcast message authentication code scheme (i.e., TESLA [1] and derivatives) because its security relies on time synchronization in the first place. Circumventing this paradox by local time synchronization between neighboring nodes requires them to transmit messages themselves. We want to avoid this because relying on neighbors creates a vulnerability against node capture attacks as one or many individual nodes on the network may be controled by an attacker.

Messages sent by the source node are authenticated by the use of digital signatures. Therefore, the nodes must be able to perform asymmetric signature verification. As the source node must transmit all synchronization messages, we are assuming a more powerful source node, e.g., a wired base station or a sensor node with additional batteries and/or a more powerful CPU.

Our performance evaluation shows that RSA is a better choice than ECDSA. In the RSA signature scheme, signature verification is faster and less energy demanding than signature creation. To speed up signature verification and to reduce the power consumption associated with it, short public exponents should be used, i.e., those with a low Hamming weight [3]. While signature creation remains quite costly in RSA, signature verification is faster than in any other well-researched, feasible signature scheme. In particular, signature verification is more costly with ECDSA than with RSA on the same security level. For equal performance, an inferior security level has to be chosen with ECDSA, i.e., on a TelosB node, 160 bit ECDSA verification is a bit slower than 2048 bit RSA verification [16], while offering a security level of 80 instead of 112 bit [2].

Asymmetric keysize must be chosen sufficiently large such that an attacker cannot obtain the private signature key. The actual keysize is left to the application developer and depends on the selection of sensor hardware and the security and lifetime requirements. Keysizes generally considered sufficiently secure through 2030, are 2048 bit for RSA and 224 bit for ECDSA [2].

Key storage is considered feasible because only a single public key must be stored. TelosB nodes offer 1 MB of data flash and 48 kB of program flash memory. Even MICA nodes offer 128 kB of program flash memory, so storing a 2048 bit RSA modulus and a public exponent is no challenge. Usually, a short verification key will be used, which can be as small as 17 bit without harming security [3], i.e., $2^{16} + 1 = 65537$.

The public key must be stored in the clients before deploying them. This can be done when first installing the software on the sensor nodes by the deployer. If the private key is not compromised, there is no need to ever change the public

network	synchronization interval			
lifetime	10 seconds	1 minute	1 hour	1 day
1 day	14	11	5	-
1 month	19	16	10	5
1 year	22	20	14	9
3 years	24	21	15	11
10 years	25	23	17	12
100 years	29	26	20	16
1000 years	32	29	24	19

Table 1: Counter size requirements

key on the sensor nodes. If the source node fails and has to be exchanged, the new source node must use the same private key as the old source node used before.

The protocol uses a continuously increasing counter for each message that is included and signed in every synchronization message. It allows for the use of more flexible data formats for the timestamps, i.e., they may repeat or decrease. In applications that use a continuously growing timestamp, the counter is not necessary, as the timestamp itself is always increasing and can thus be used as counter. However, the counter is strictly required for applications of the protocol for other purposes than time synchronization. The counter must be of sufficient size such that it will not wrap around during the lifetime of the network. E.g., if time synchronization is done every minute, a 24 bit counter would last longer than 10 years (see Table 1). The clients will only accept messages with a counter value greater than their internal counters.

The internal counter must not be changed by the client before verifying the signature. Otherwise, an attacker could increase the node's counter, thus preventing the node from synchronizing again in the near future. The internal counter must not be changed after the client has set its clock, but right before it. Otherwise, an attacker with the ability to abort the operation, e.g., by causing an interrupt or restarting the node, would be able to replay the last message. Therefore, the only correct order of action for the receiver regarding the counter corresponds to the steps 4)–7) in the protocol description.

The counter starts at 0 for both receivers and source nodes when they are newly deployed at the same time. The counter may be set to 0 in new stations if they are to be added to a network where the presence of an attacker can be excluded until the next synchronization happens. If stations are to be added in a hostile environment, the counter must be set to a value slightly larger than the one in use at the time of deployment. If the source node fails and has to be exchanged the counter on the new source node must start at a value at least as big as the last one used.

4.2 Security Properties

In this section, we clarify the attacker model and examine how our protocol prevents certain attacks.

We assume an adversary model where the attacker is able to receive all traffic, to transmit messages with faked sender identity and to block the channel. Furthermore, we assume that the source node cannot be captured by an attacker, either because it is physically protected or because it is not in the hostile environment.

In a *node capture attack*, the attacker is able to read all data stored on a node and possibly change its behavior. In the proposed protocol, no secret information can be extracted from individual nodes. Also, individual nodes are not required for a successful protocol run of any other node. Nodes trying to impersonate the source node cannot create correct message signatures, so even a single remaining node will be able to synchronize to the source node. Thus, it is secure against node capture as long as the source node remains uncompromised.

In a *man-in-the-middle-attack*, a node alters the content of the message while relaying it. Our protocol is not vulnerable to such an attack, as the signature verification will fail on the receiver's side.

In a *wormhole attack*, an attacker creates a physical link between two distant nodes on the network that has high bandwidth and low delay. Thus, neighboring nodes adjust their routing such that many packets travel through the new link. The link may be used for man-in-the-middle-attacks. *Byzantine traitors* is an attack where the neighbors of a node communicate contradicting information to it and the node must determine which information is true. Typically, such an attack could be executed after a number of nodes were captured. Neither wormhole nor Byzantine traitor attacks apply to our protocol because there is no communication or routing between individual nodes.

An attacker might introduce arbitrary packets with valid counters and random signature data during a synchronization interval without obstructing the legit packet. The receiver would then have to determine which packet is the valid one through multiple signature verifications. This results in an exhaustion of resources for the node. The effect of the attack can be shifted from lifetime reduction to reduced clock precision by verifying only one signature per synchronization interval by choice of the deployer.

The protocol cannot provide protection against an attacker blocking the channel. However, it can recover from a channel blocked over a finite amount of time.

The counter in the protocol provides protection against an attacker replaying old messages from previous synchronization runs (i.e., setting the clock back). If the station has already received a newer message, it will not accept old messages.

An attacker that is able to block radio reception of a sensor node over a sufficient period of time would be able to make the sensor node accept old valid messages sent after the start of the radio blocking. This is called a *pulse delay attack*. With the use of filters on the nodes, some protection against pulse delay attacks can be achieved. Filtering means rejecting synchronization messages if the new timestamp differs too much from the local time, i.e., if δ_{clock} exceeds a threshold value. Using filters is only possible in applications that use a continuous time format. As the nodes are only listening for synchronization messages when they are scheduled, a pulse delay attack would either introduce a clock offset δ_{clock} of roughly the size of a synchronization interval or a multiple of it, which is easy to detect because δ_{clock} will be suspiciously large, or it will be smaller than the receiver's radio reception window. Thus, we need to define a threshold value that establishes an upper bound on δ_{clock}. As filters are a countermeasure that is based on time synchronization itself they shall be handled with care, i.e., the threshold shall be determined after a thorough evaluation of the clock offsets experienced under similar conditions without an attacker. Even without filters, the node's clock will return to the authentic time at the next synchronization after the channel is unblocked.

Not requiring the receiving nodes to transmit any messages for our protocol achieves a number of desirable goals: (1) An attacker is unable to locate or count the nodes. (2) There is no forwarding of packets that could decrease the precision of the synchronization or reduce the sensor nodes' battery life. (3) The source node does not need to be in the transmission range of the receiving nodes. Therefore, if the source node is capable of a greater signal power, it may be placed at a distance to the receiving nodes. This is particularly interesting if the network itself is located inside a hostile environment, as then the source node may still be located outside this environment (see Figure 1). Directional antennas may further support this.

Although we do not require a specific timestamp format, the effective use of filters to limit pulse delay attacks depends on the use of a continuous time format. When continuous time formats are used, our protocol resists more attacks than other protocols [6, 8, 10] and allows nodes to skip synchronization phases to conserve energy. With a non-continuous time format, pulse delay attacks cannot be defeated. As other protocols do not allow non-continuous time formats, this seems acceptable.

5 Performance of the new Protocol

5.1 Performance Model

We compare the performance of the new protocol to TinySeRSync regarding energy cost and node workload. We chose TinySeRSync because it comes closest to our protocol in terms of security. An implementation of the protocol is freely available [17]. We used it to derive the size of the messages exchanged between the nodes. Our analysis is based on a number of assumptions:

- All nodes are able to access the channel immediately, i.e., there is no waiting for a free time slot.
- There are no transmission errors or collisions.
- Filtering to prevent pulse delay attacks takes no time or energy.

operation	energy [mJ]	time [ms]
transmit 1 bit	0.000153	0.04
receive 1 bit	0.000226	0.04
verify RSA 1024	2.700	220.0
verify RSA 2048	12.20	1000
verify ECDSA 160	12.41	1020
verify ECDSA 224	33.55	2760

Table 2: Cost of operations on TelosB platform [16]

- Calculating and verifying MACs or a hash function takes no time or energy.
- Waking up from sleep mode or returning to it takes no time or energy.
- Messages are not padded or split into different packets.
- We do not consider the cost of the source node in our protocol, as it is a base station.

These assumptions make communication less costly, so the results are influenced in favor of the TinySeRSync protocol.

We refer to ,,Global synchronization“ by all stations synchronizing to a source node at the same time. Costs are estimated for a single node in one global synchronization interval, i.e., the period between the end of one global synchronization and the end of the next global synchronization. In the following, all computationally expensive tasks and protocol messages are discussed.

The same data types as in the TinySeRSync implementation [17] are used: A 16 bit Node ID and a 32 bit timestamp. In addition, we use a 32 bit update counter for our protocol.

The overhead per packet in TinyOS, i.e., the size of all headers and trailers, was estimated as 288 bits [7]. This value is used in the following calculations.

Our protocol: A source node directly broadcasts a global synchronization message m to all stations. Each node must receive the message m once per global synchronization interval. The size of m and the effort associated with it is as follows:

- The message m contains ID, timestamp, counter, and signature. The total size of m including the overhead is 368 bits plus the size of the signature (depending on the signature scheme used, from 320 to 2048 bits).
- Each node has to verify a signature by calculating a hash function on a 64 bit input and has to perform a cipher-dependent public key operation.
- No message is sent by the nodes.

TinySeRSync: Let n denote the number of nodes of each neighbor and d denote the number of pairwise synchronizations per global synchronization.

Pairwise synchronization: In each of the d pairwise synchronizations between two global synchronizations each node sends a request message p_1 to all of its neighbors. This message includes the ID and a timestamp. The total size of p_1 is 480 bits [17]. Each neighbor answers with a response message p_2, which includes the IDs and three timestamps. The total size of p_2 is 544 bits [17]. As a consequence, in each pairwise synchronization round, each node sends and receives $(480 + 544)n = 1024n$ bits.

Global synchronization: The global synchronization message g_1 is protected with μTESLA. The total size of g_1 is 576 bits [17]. The key is disclosed with key disclosure message g_2 shortly after g_1 was sent. The total size of g_2 is 352 bits [17]. Both messages are rebroadcasted by all other nodes. Thus during global synchronization each node sends g_1 and g_2 once and receives g_1 and g_2 n times. This amounts to sending $576+352 = 928$ bits and receiving $(576+352)n = 928n$ bits in each global synchronization round.

The total amount of bits sent (s) and received (r) by each node during one global synchronization interval is summarized in (1) and (2), where $|x|$ denotes the bit length of message x:

$$\begin{aligned} s &= (n \cdot |p_1| + n \cdot |p_2|) \cdot d + |g_1| + |g_2| && (1)\\ &= (n \cdot 480 + n \cdot 544) \cdot d + 576 + 332 \\ &= 1024nd + 928 \\ r &= (n \cdot |p_1| + n \cdot |p_2|) \cdot d + n \cdot |g_1| + n \cdot |g_2| && (2)\\ &= (n \cdot 480 + n \cdot 544) \cdot d + n \cdot 576 + n \cdot 352 \\ &= 1024nd + 928n \end{aligned}$$

For the cost of public key operations on the TelosB platform, as well as transmission and reception, we rely on the data presented in [16] (see Table 2).

Protocol	neighbors n	energy [mJ]	time [ms]
RSA 1024	any	3.01	225.6
RSA 2048	any	12.75	1009.7
ECDSA 160	any	12.57	1022.8
ECDSA 224	any	33.73	2763.3
TinySeRSync (d=2.5)	2	2.50	52.1
TinySeRSync (d=2.5)	6	7.22	148.9
TinySeRSync (d=2.5)	10	11.94	245.6
TinySeRSync (d=2.5)	14	16.66	342.4
TinySeRSync (d=20)	2	16.09	338.8
TinySeRSync (d=20)	6	47.97	1009.0
TinySeRSync (d=20)	10	79.86	1679.2
TinySeRSync (d=20)	14	111.75	2349.4

Table 3: Cost of global synchronization on TelosB platform per node

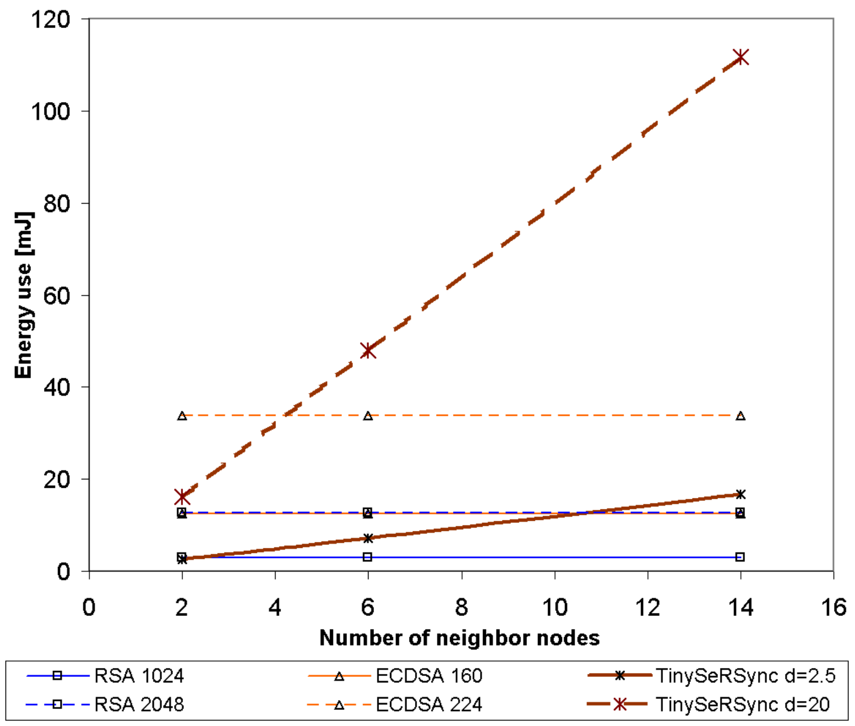

Figure 3: Engery use [per node and global sync] depending on the number of neighbors

We are assuming an average transmission level of the TelosB. Other authors estimate the cost of communication up to five times higher [13], but to again favor TinySeRSync we chose the lower values.

With Table 2, (1), and (2) we can now compare our protocol to TinySeR-Sync in terms of energy use and workload per node per global synchronization interval. In the original paper [10], the number d of pairwise synchronizations per global synchronization is 2.5. In the implementation, it is set to $d = 20$. Because the size of d has a strong influence on the performance, we compare our protocol to both values of d. Note that our protocol does not depend on the number of neighbor nodes. The results are shown in (Table 3), (Figure 3), and (Figure 4). Note that RSA 2048 and ECDSA 160 overlap because they are almost equally expensive in terms of energy and workload.

5.2 Results for Energy Consumption (Figure 3)

Considering $d = 2.5$, our protocol with RSA 1024 is the most energy efficient solution for any $n \geq 3$. RSA 1024, ECDSA 160, and RSA 2048 are superior to TinySeRSync for $d = 20$ and any n. ECDSA 224 outperforms TinySeRSync only for $n \geq 4$ and $d = 20$.

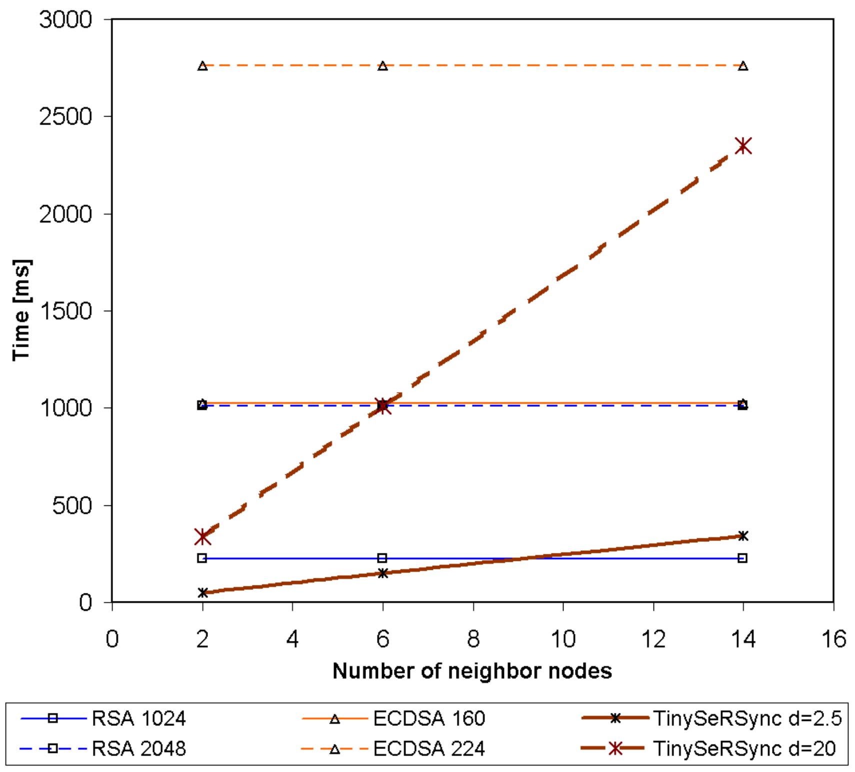

Figure 4: time [per node and global sync] depending on the number of neighbors

5.3 Results for Time Consumption (Figure 4)

Considering workload, TinySeRSync is the fastest solution for $d = 2.5, n < 9$. RSA 1024 is faster than TinySeRSync with $d = 20$ for any n. ECDSA 160 and RSA 2048 are faster than TinySeRSync with $d = 20, n \geq 6$. ECDSA 224 is the slowest solution.

5.4 Discussion

The performance of TinySeRSync depends largely on the density of the network, i.e., the amount of neighbors the pairwise synchronization is done with (n) and how often it is performed per global synchronization (d). Our protocol is independent of any such variables and will not cause additional workload or energy use on the nodes for any network density. With RSA 1024, our protocol is more energy efficient than TinySeRSync and still feasible within 225.6 ms per synchronization. ECDSA 160 and RSA 2048 perform almost identically, using less energy than TinySeRSync for growing values of d and dense networks. However, ECDSA 160 and RSA 2048 take more time in most combinations of d and n. ECDSA 224 requires 2763 ms per synchronization and will use more energy than TinySeRSync in most combinations of d and n.

Our simulation depends on assumptions and energy usage data that both favor TinySeRSync. We are expecting even better results from measurements of an implementation.

6 Deployment Options and Optimizations

Our protocol allows for a lot of tuning to the special needs of individual applications, even beyond wireless sensor networks. In applications that focus on data gathering with later offline evaluation, pulse delay attacks can be countered in another way than filters. The nodes shall record the counter value of every accepted message together with their old internal time. The source shall keep track of the messages it sent. After the sensor nodes (or the data on them) are retrieved by the deployer, the log files can be used to calculate the true time of the events recorded or at least to detect pulse delay attacks.

With continuous timestamps and a filter, the nodes may be allowed to decide themselves if they need synchronization. This decision can be based on the clock skew experienced earlier, e.g., by ignoring one more synchronization message than before if the clock offset is below a threshold value, and one less (or none) if it is above another threshold value. Not synchronizing to every message achieves better energy efficiency at the cost of synchronization precision. Nodes must listen to and verify all synchronization messages in applications with non-continuous timestamps because *tsp* values can be unpredictable for sensor nodes.

The time between two synchronization messages must be greater than the time required for the signature verification, and long enough not to clog the channel and not to drain the batteries of the receiving nodes if they cannot decide themselves whether to synchronize or not. Influencing factors are the accuracy desired, the accuracy of the nodes' internal clocks, the verification speed, and lifetime considerations.

The communication channel used for transmitting the synchronization messages can either be a dedicated channel not used for other applications, or the regular communication channel between the nodes. Dedicated channels may be used if special hardware is available or if there is no communication between the nodes, e.g., radio controlled clocks in need of security. Message transmission over a regular communication channel requires a form of collision avoidance so that individual inter-node communication will not interfere with time synchronization. As we cannot rely on time synchronization and because there is no channel back to the source node, it must transmit a preamble at the start of each time synchronization message long enough so that other nodes trying to send at the same time will stop sending. Individual nodes must detect this preamble and reschedule their communication.

As a dedicated channel, low/medium frequencies may be used because of the increased range and obstacle penetration.

7 Conclusion

We have presented a protocol that allows for replay protected authentic distribution of timestamps and other payloads in a one-way broadcast communication scenario. The protocol assumes the existence of a single source node that may not be captured, that is able to create signatures, and that can transmit messages to all stations on the network. There is no need for communication between the nodes for time synchronization purposes with the proposed protocol. An attacker observing the radio channel during synchronization will not learn anything about the network, which is unique for a time synchronization protocol in sensor networks.

With a continuous time format, our protocol resists the same attacks as other protocols [6, 8, 10], and additionally resists any number of compromised nodes. With a non-continuous time format, pulse delay attacks cannot be detected. We have shown that our protocol is energy efficient when compared to another secure time synchronization protocol and causes a tolerable amount of workload on the sensor nodes.

Mobility is supported as long as the nodes remain inside the source node's transmission range. The proposed protocol also scales very well to large networks, because no additional messages must be sent when new nodes join a network. If the spatial size of the network increases, the source node's transmission power may need to be adjusted.

8 Future Work

The implementation on TelosB and possibly other nodes will help us determine the precise energy cost and workload of our proposed protocol, especially when compared to existing protocols. With a state-of-the-art cross-layer implementation, we expect our protocol to perform better than in the simulation. We also expect to perform well in terms of precision, as all nodes perform the same operation on the same hardware at virtually the same time.

In addition, increasing the possible spatial network size by using multiple source nodes in different locations on the network transmitting at different times will be evaluated.

Acknowledgments

This work has been supported by the UMIC Research Centre, RWTH Aachen University.

References

[1] J.D. Tygar Dawn Song Adrian Perrig, Ran Canetti. The TESLA broadcast authentication protocol. *CryptoBytes*, 5:2, 2002.

[2] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for key management - part 1: General. In *NIST Special Publication 800-57, August 2005, National Institute of Standards and Technology*, 2005.

[3] D. Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of the American Mathematical Society (AMS)*, 46 No 2, 1999.

[4] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. In *OSDI '02: Proceedings of the 5th symposium on operating systems design and implementation.*

[5] Saurabh Ganeriwal, Ram Kumar, and Mani B. Srivastava. Timing-sync protocol for sensor networks. In *SenSys '03: Proceedings of the 1st international conference on embedded networked sensor systems.*

[6] Ganeriwal, Srivastava. Secure time synchronization in sensor networks. *ACM transactions on information and systems security*, 2008.

[7] E.; Sadok D. Guimaraes, G.; Souto and J Kelner. Evaluation of security mechanisms in wireless sensor networks. In *Systems Communications Proceedings*, 2005.

[8] Mi Wen Hui Li, Yanfei Zheng and Kefei Chen. *A Secure Time Synchronization Protocol for Sensor Network in Lecture Notes in Computer Science.* Springer Berlin / Heidelberg, 2007.

[9] Joengmin Hwang and Yongdae Kim. Revisiting random key predistribution schemes for wireless sensor networks. In *SASN '04: Proceedings of the 2nd ACM workshop on security of ad hoc and sensor networks*, 2004.

[10] Kun Sun, Peng Ning, Cliff Wang, An Liu, Yuzheng Zhou. TinySeRSync: Secure and resilient time synchronization in wireless sensor networks. In *Proceedings of the 13th ACM conference on computer and communications security*, 2006.

[11] Michael Manzo, Tanya Roosta, and Shankar Sastry. Time synchronization attacks in sensor networks. In *SASN '05: Proceedings of the 3rd ACM workshop on security of ad hoc and sensor networks*, 2005.

[12] Maróti, Kusy, Simon, and Lédeczi. The flooding time synchronization protocol. In *SenSys '04: Proceedings of the 2nd international conference on embedded networked sensor systems.*

[13] Standaert Meulenaer, Gosset and Pereira. On the energy cost of communication and cryptography in wireless sensor networks.

[14] Mills. DCNET Internet Clock Service (RFC 778). IETF, 1981.

[15] Wen Culler Perrig, Szewczyk and Tygar. Spins: Security protocols for sensor networks. In *Wireless Networks*, 2001.

[16] Peter Piotrowski, Langendoerfer. How public key cryptography influences wireless sensor node lifetime. In *Proceedings of the fourth ACM workshop on security of ad hoc and sensor networks*, 2006.

[17] Sun, Dr. Ning, Dr. Wang, Liu, Zhou, Kampanakis. Tinysersync: Secure and resilient time synchronization for wireless sensor networks (version 0.1), 2006.

[18] Yin, Qi, Fu. ASTS: An agile secure time synchronization protocol for wireless sensor networks. In *Wireless Communications, Networking and Mobile Computing*, 2007.