TinySeRSync: Secure and Resilient Time Synchronization in Wireless Sensor Networks[*]

Kun Sun, Peng Ning
Computer Science Dept.
NC State University
Raleigh, NC 27695
{ksun3,pning}@ncsu.edu

Cliff Wang
Army Research Office
RTP, NC 27709
cliff.wang@us.army.mil

An Liu, Yuzheng Zhou
Computer Science Dept.
NC State University
Raleigh, NC 27695
{aliu3,yzhou3}@ncsu.edu

ABSTRACT

Accurate and synchronized time is crucial in many sensor network applications due to the need for consistent distributed sensing and coordination. In hostile environments where an adversary may attack the networks and/or the applications through external or compromised nodes, time synchronization becomes an attractive target due to its importance. This paper describes the design, implementation, and evaluation of TinySeRSync, a secure and resilient time synchronization subsystem for wireless sensor networks running TinyOS. This paper makes three contributions: First, it develops a *secure single-hop pairwise time synchronization* technique using *hardware-assisted, authenticated medium access control (MAC) layer timestamping.* Unlike the previous attempts, this technique can handle high data rate such as those produced by MICAz motes (in contrast to those by MICA2 motes). Second, this paper develops a *secure and resilient global time synchronization* protocol based on a novel use of the μTESLA broadcast authentication protocol for *local authenticated broadcast*, resolving the conflict between the goal of achieving time synchronization with μTESLA-based broadcast authentication and the fact that μTESLA requires loose time synchronization. The resulting protocol is secure against external attacks and resilient against compromised nodes. The third contribution consists of an implementation of the proposed techniques on MICAz motes running TinyOS and a thorough evaluation through field experiments in a network of 60 MICAz motes.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Wireless communication*

[*]This work is supported by the National Science Foundation (NSF) under grant CAREER-0447761. Wang's work is supported by the US Army Research Office (ARO) under staff research grant W911NF-04-D-0003-0001.

General Terms

Security, Design, Algorithms

Keywords

Sensor Networks, Security, Time Synchronization

1. INTRODUCTION

Recent technological advances have made it possible to develop distributed sensor networks consisting of a large number of low-cost, low-power, and multi-functional sensor nodes that communicate over short distances through wireless links [6]. Such sensor networks are ideal candidates for a wide range of applications such as monitoring of critical infrastructures, data acquisition in hazardous environments, and military operations. The desirable features of distributed sensor networks have attracted many researchers to develop protocols and algorithms that can fulfill the requirements of these applications (e.g., [6,14,16,21,30,31,36]).

Accurate and synchronized time is crucial in many sensor network applications, particularly due to the need for consistent distributed sensing and coordination. However, due to the resource constraints on typical sensor nodes such as MICA motes [8], traditional time synchronization protocols (e.g., NTP [27]) cannot be directly applied in sensor networks.

A number of time synchronization protocols (e.g., [11,13, 17, 23, 26, 28, 33, 37]) have been proposed for sensor networks to achieve *pairwise* and/or *global* time synchronization. Pairwise time synchronization aims to establish relative clock offsets between pairs of sensor nodes, while global time synchronization aims to provide a network-wide time reference for all the sensor nodes in a network. Existing pairwise or global time synchronization techniques are all based on *single-hop* pairwise time synchronization, which discovers the clock difference between two neighbor nodes that can communicate with each other directly. Two approaches have been used for single-hop pairwise clock synchronization: *receiver-receiver synchronization* (e.g., RBS [11]), in which a reference node broadcasts a reference packet to help a pair of receivers identify their clock differences, or *sender-receiver synchronization* (e.g., TPSN [13]), where a sender communicates with a receiver to estimate their clock difference. Multi-hop pairwise clock synchronization protocols and most of the global clock synchronization protocols (e.g., [11, 13, 37]) establish multi-hop paths in a sensor network, so that all the nodes in the network can synchronize their clocks to the source based on the single-hop pairwise clock differences between adjacent nodes in these paths. Alternatively, diffusion based global synchronization protocols [23] have the nodes' clocks converge by spreading synchronization information locally.

1.1 Threats to Time Synchronization in Wireless Sensor Networks

Most of existing time synchronization techniques developed for sensor networks assume benign environments. However, in hostile environments, an adversary may certainly attack the time synchronization protocol due to its importance. Note that all time synchronization protocols rely on *time-sensitive* message exchanges. To mislead these protocols, the adversary may forge and modify time synchronization messages, jam the communication channel to launch Denial of Service (DoS) attacks, and launch pulse-delay attacks [12] by first jamming the receipt of time synchronization messages and then later replaying buffered copies of these messages. The adversary may also launch wormhole attacks [18] by creating low latency and high bandwidth communication channels between different locations in the network, and (selectively) delay or drop time synchronization messages transmitted through the wormholes. The adversary may use Sybil attacks [10, 29], where one node presents multiple identities, to defeat typical fault tolerant mechanisms. Though message authentication can be used to validate message sources and contents, it cannot validate the *timeliness* of messages, and thus is unable to defend against all of these attacks.

Moreover, the adversary may compromise some nodes, and exploit the compromised nodes in arbitrary ways to attack time synchronization. For example, the adversary may instruct the compromised nodes to (selectively) delay or drop time synchronization messages, and launch Sybil attacks [29] using the identities and keying materials of compromised nodes if message authentication is enabled. The adversary may also instruct the compromised nodes not to cooperate with others, and inject false time synchronization messages. The compromised nodes may collude with each other to cause the worst damage to the network.

1.2 Inadequacy of Current Solutions

It is natural to consider fault-tolerant time synchronization techniques, which have been studied extensively in the context of distributed systems (e.g., [7, 9, 22, 32, 39]). However, these techniques require either digital signatures (e.g., HSSD [9], CSM [22]), exponential copies of messages (e.g., COM [22]), or a completely connected network (e.g., CNV [22]) to prevent malicious nodes from modifying or destroying clock information sent by normal nodes. Thus, they are impractical for resource-constrained sensor nodes. A recent work provides an efficient fault-tolerant time synchronization protocol for a cluster of fully connected nodes by exploiting the broadcast nature of wireless communication [40]. However, it requires a trusted entity during network initialization to avoid heavy communication overhead, and does not allow incremental deployment of additional sensor nodes.

There have been several recent studies for secure and resilient time synchronization in sensor networks [12, 25, 38, 41]. Ganeriwal et al. proposed several techniques for secure pairwise synchronization (SPS), secure multi-hop synchronization, and group-wise synchronization [12]. The SPS technique provides authentication for medium access control (MAC) layer timestamping by adding timestamp and message integrity code (MIC) as the messages being transmitted. This approach works for low data rate sensor radios (e.g., CC1000 on MICA2 motes with 38.4Kbps data rate); however, it cannot keep up with recent IEEE 802.15.4 [20] compliant sensor radios such as CC2420 on MICAz and TelosB, whose data rate is 250Kbps. The group synchronization in [12] uses pairwise authentication to synchronize a group of nodes, thus introducing high computation and communication overheads. Moreover, the group synchronization assumes all nodes in a group can communicate with each other directly. Extensions to groups with multi-hops are speculated, but no specific solution is provided.

Manzo et al. discussed a few attacks against existing time synchronization protocols and several countermeasures to protect single-hop and multi-hop time synchronization [25]. However, there was no mechanism to authenticate the timeliness of synchronization messages, and thus no protection against, for example, pulse-delay attacks [12] and worm-hole attacks [19], in which the adversary may delay authenticated synchronization messages. Moreover, though μTESLA was suggested as a way to authenticate broadcast synchronization messages, no solution was given to resolve the conflict between the goal of achieving time synchronization and the fact that μTESLA requires loose time synchronization.

Sun et al. proposed a resilient time synchronization protocol that can deal with various attacks including compromised nodes [41]. However, similar to [25], the proposed techniques cannot authenticate the timeliness of synchronization messages, thus suffering from pulse-delay [12] and wormhole attacks [19]. In addition, the approaches in [41] use authenticated unicast communication to propagate global synchronization messages. This introduces substantial communication overhead as well as frequent message collisions in dense sensor networks [41].

Song et al. investigated countermeasures against attacks that mislead sensor network time synchronization by delaying synchronization messages [38]. They proposed two methods for detecting and tolerating delay attacks: one transforms attack detection into statistical outliers detection, and the other detects attacks by deriving the bound of the time difference between two nodes through message exchanges. Unfortunately, [38] only addresses synchronization of neighbor nodes, but does not support global time synchronization in multi-hop sensor networks.

1.3 Our Contributions

To address the aforementioned problems, we develop a secure and resilient time synchronization subsystem called *TinySeRSync* for wireless sensor networks, targeting common sensor platforms such as MICAz and TelosB running TinyOS [16]. Our solution offers a novel way to integrate (broadcast) authentication into time synchronization, which successfully provides authentication of the source, the content, and the timeliness of synchronization messages. Our solution not only addresses secure time synchronization between neighbor nodes, but also the global synchronization of an entire sensor network.

We make three contributions in this paper:

1. We develop a *secure single-hop pairwise time synchronization* technique using *hardware-assisted, authenticated medium access control (MAC) layer timestamping.* Unlike the previous attempts, this technique can

handle high data rate such as those produced by MICAz and TelosB motes (in contrast to those by MICA2 and MICA2DOT motes).

2. We develop a *secure and resilient global time synchronization* protocol based on a novel use of the μTESLA broadcast authentication protocol for *local authenticated broadcast*, resolving the conflict between the goal of achieving time synchronization with μTESLA-based broadcast authentication and the fact that μTESLA requires loose time synchronization. The resulting protocol is secure against external attacks and resilient against compromised nodes.

3. We provide an implementation of the proposed techniques on TinyOS and a thorough evaluation through field experiments in a network of 60 MICAz motes. The evaluation results indicate that TinySeRSync is a practical system for secure and resilient time synchronization in wireless sensor networks.

1.4 Organization of the Paper

The rest of the paper is organized as follows. The next section gives an overview of the TinySeRSync system. Section 3 and Section 4 describe the secure pairwise time synchronization and the secure and resilient global time synchronization in TinySeRSync, respectively. Section 5 provides the security and performance analysis of TinySeRSync. Section 6 discusses a few implementation issues. Section 7 presents the experimental evaluation of TinySeRSync in a network of 60 MICAz motes. Section 8 concludes this paper and points out future research directions.

2. OVERVIEW OF PROPOSED APPROACH

We assume that a sensor network consists of a large number of resource constrained motes such as MICA series of motes. We assume there is a *source node* S that is well synchronized to the external clock, for example, through a GPS receiver. We would like to synchronize the clocks of all the sensor nodes in the network to that of the source node. We assume the source node is trusted, and all the other nodes know the identity of the source node. The assumption of the single, trusted source node is to simplify the discussion in this paper. Our approach can be easily modified to accommodate multiple source nodes in order to enhance the performance, improve the availability of source nodes, and/or tolerate potentially compromised source nodes.

To deal with the ad hoc deployments of sensor networks and the lack of initial synchronization among sensor nodes, we propose to achieve global time synchronization in a sensor network in two *asynchronous* phases: *Phase I–secure single-hop pairwise synchronization*, and *Phase II–secure and resilient global synchronization*. In Phase I, pairs of neighbor nodes exchange messages with each other to obtain single-hop pairwise time synchronization. Phase I uses authenticated MAC layer timestamping and a two message exchange to ensure the authentication of the source, the content, and the timeliness of synchronization messages. Nodes run Phase I periodically to compensate (continuous) clock drifts and maintain certain pairwise synchronization precision, providing the foundation for global time synchronization as well as the μTESLA-based local broadcast authentication in Phase II.

Phase II uses *authenticated local (re)broadcast* to achieve global time synchronization, starting with a broadcast synchronization message from the source node. Phase II adapts μTESLA to ensure the timeliness and the authenticity of the local broadcast synchronization messages. To be resilient against potential compromised nodes, each node estimates multiple candidates of the global clock using synchronization messages received from multiple neighbor nodes, and chooses the median. Nodes that are synchronized to the source node further rebroadcast the synchronization messages locally. This process continues until all the nodes are synchronized. Phase II also runs periodically to maintain certain global time synchronization precision.

We would like to emphasize that the two phases are *asynchronous*. In other words, secure single-hop pairwise synchronization (Phase I) is executed by nodes individually and independently, while secure and resilient global synchronization (Phase II) is controlled by the source node and propagated throughout the network. The only requirement is that a node finishes Phase I before entering Phase II. Also note that both Phase I and Phase II are executed periodically. Though a node that has not performed Phase I synchronization with its neighbor nodes cannot participate in a global synchronization, it may join the next round of global synchronization once it finishes Phase I. Thus, our approach supports incremental deployment of sensor nodes, which is an important property required by many sensor network applications.

We present the two phases of TinySeRSync in detail in the next two sections.

3. PHASE I: SECURE SINGLE-HOP PAIRWISE TIME SYNCHRONIZATION

The goal of secure single-hop pairwise time synchronization is to ensure two neighbor nodes can obtain their clock difference through message exchanges in a secure fashion. This requires the authentication of the source, the content (i.e., the timing information), and the timeliness of each message used for such synchronization.

In the following, we first discuss how we provide authentication of the source and the timing information in synchronization messages, and then describe a secure two-way pairwise time synchronization protocol for a node to obtain the clock difference from a neighbor node.

3.1 Authenticated MAC Layer Timestamping

MAC layer timestamping has been widely accepted as an effective way to reduce the synchronization error during the message exchanges since it was proposed in [13]. By adding (on the sender's side) and retrieving (on the receiver's side) timestamps in the MAC layer, this approach avoids the uncertain delays introduced by application programs and medium access, and thus has more accurate synchronization precision.

To ensure the integrity of pairwise time synchronization, we may authenticate a synchronization message by adding a MIC once the MAC layer timestamp is added, assuming the two nodes performing pairwise synchronization share a secret pairwise key through, for example, TinyKeyMan [24]. This, however, introduces a potential problem due to the extra delay required by the MIC generation: It is necessary to have a MAC layer timestamp that marks the exact transmission time of a certain bit in the message at the sender's side, but the MIC generation and insertion require extra delay and have to be done after the timestamp is inserted into the message.

The delay introduced by MIC generation and insertion can be tolerated for sensor platforms with low data rate radio components, such as MICA2 motes. In an earlier study [12], Ganeriwal et al. attempted to provide authenticated MAC layer timestamping for MICA2 motes (38.4 kbps data rate) by generating MIC on the fly. Specifically, when the radio component of a sensor node begins to transmit the first byte of a synchronization message, it appends the current timestamp into the message, calculates the MIC, and appends the MIC into the message being transmitted. Due to the low data rate (38.4 kbps), the MAC layer timestamp and the MIC can be added into the packet before the corresponding bytes are transmitted [12]. However, with the increased data rate on recent sensor platforms with IEEE 802.15.4 compliant radio components (250 kbps data rate [20]), such as MICAz and TelosB motes, there is not enough time to generate and insert the MIC before the transmission of the MIC bytes due to the delay introduced by the MIC calculation [12].

We propose a prediction-based approach to address the above problem. In the following, we describe our approach, with a specific target of the IEEE 802.15.4 compliant radio component ChipCon CC2420 [3], which is commonly used in recent sensor platforms such as MICAz and TelosB motes. We also assume the sensor nodes use TinyOS [16], the open source operating system for networked sensor nodes.

3.1.1 Prediction-Based MAC Layer Timestamping and Hardware-Assisted Authentication

We observe that the code for generating a MIC is deterministic, and the time required for a MIC generation for messages with a given length (or, more precisely, a given number of blocks) is fixed. In addition, the process to transmit a packet (starting from observing the channel vacancy to the actual transmission of data payload) in CC2420 is also deterministic. Thus, when we put a timestamp into a synchronization message to be authenticated in the MAC layer, we may predict the time required by MIC generation and at the same time predict the delay between the start of transmission and the transmission a given bit in the packet.

Let us review how a sensor node (such as a MICAz mote) equipped with a CC2420 radio component handles packet transmission on TinyOS. Figure 1 shows the transmission and receiving process. When a node has a message to send, its micro-controller first transmits the message to the RAM (TXFIFO buffer) of the CC2420 radio component. After the buffering is done, CC2420 sends a signal to the micro-controller. At this time, if the radio channel is clear, the micro-controller signals CC2420 to send out the packet with a STXON strobe. Otherwise, it will back off randomly and then test the channel again. After receiving a STXON signal, CC2420 first sends 12 symbol periods, with 4 bits in each symbol, and then sends 4 byte preamble and 1 byte of Start of Frame Delimiter (SFD) field, followed by 1 byte length field and the MAC Protocol Data Unit (MPDU). The sequence of events follows strict timing, and the delays introduced by all of them are predictable.

We use the last bit of the SFD byte as the reference point for time synchronization. In other words, the sender takes the transmission (completion) time of the last bit of SFD as the MAC layer transmission timestamp, and the receiver marks the receiving time of the same bit as the receiving timestamp. To allow the sender to perform MAC layer timestamping and authentication, as mentioned earlier, we can predict the time when the last bit of SFD will be transmitted.

Sender Side: Now let us describe our proposed sending process in detail. Assume the sender has started sending a synchronization message to the RAM (TXFIFO buffer) of CC2420. At this time, the timestamp field in the message has not been filled. Upon completion of the transfer, CC2420 sends a signal to the micro-controller, which then starts handling the signal in the MAC layer. If the radio channel is clear, the micro-controller generates a timestamp by adding the current time with a constant offset Δ. This constant offset Δ is the time delay from checking the current time to the transmission of the last bit of SFD. The micro-controller then writes the timestamp directly to the corresponding bytes in CC2420's TXFIFO. Next, if the radio channel is still clear, it signals CC2420 to send out the message with a STXON strobe. Otherwise, it backs off for a random period of time and then repeats the above process. (Note that this back-off will force the micro-controller to write the MAC layer timestamp again when the same message is to be re-transmitted.) Upon receiving the STXON signal, as described earlier, CC2420 starts transmitting the symbol periods, the preamble, the SFD, and the MPDU. In the case when CC2420 can successfully transmit the packet, the execution and the data transmission are both deterministic, and the delay Δ is a constant. The delay Δ we obtained on MICAz motes is 399.28 μs. This includes the total transmission time for the 12 symbol periods, preamble and SFD ($(12 * 4 + (4 + 1) * 8)/250,000 = 0.000352$ s $= 352$ μs) and the execution time between checking the timestamp and starting the transmission (47.28 μs).

In our implementation, we have CC2420 start the inline authentication to generate the MIC of the message at the time when it begins to transmit the symbol periods. According to the manual of CC2420 [3], the inline authentication component in CC2420 can generate a 12-byte MIC on a 98-byte message in 99 μs. Thus, we can easily see the MIC generation can be completed before it is transmitted. Besides the MIC, CC2420 also generates a 2-byte Frame Check Sequence (FCS) using Cyclic Redundancy Check (CRC).

Receiver Side: After an approximately 2 μs propagation delay [3], the radio component CC2420 on the receiver node will receive the preamble of an incoming message. Once the SFD field is completely received by CC2420, the SFD pin will go high to signal the micro-controller, which then records the current time as the receiving timestamp. When the FIFOP pin goes low, the micro-controller will be signaled to read the data from CC2420's RXFIFO buffer, in which the first byte indicates the length of the message. During the receiving process, CC2420 performs inline verification of the MIC (and the CRC) in the message, using the pairwise key shared between the sender and the receiver. The micro-controller examines the verification result, and copies the whole packet if the packet is authenticated. All these operations are performed in the MAC layer, and transparent to the application layer.

Unlike the deterministic delay on the sender's side, the delay affecting the receiving timestamps on the receiver's side

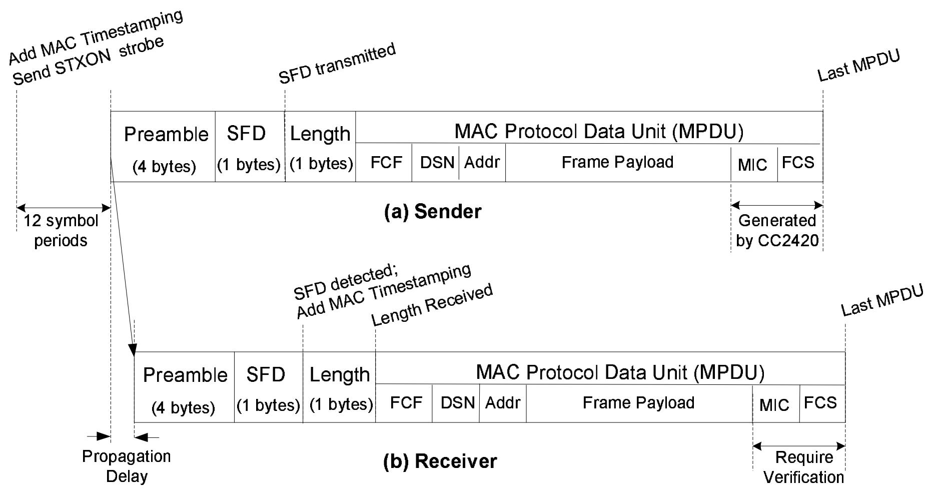


Figure 1: Packet Sending and Receiving Process

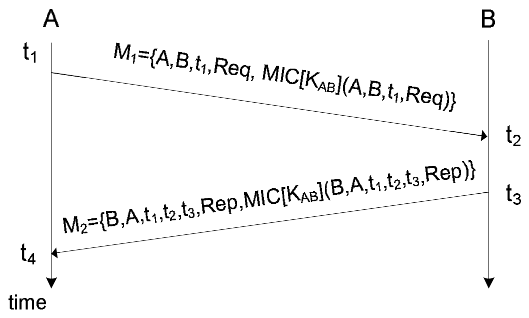

A
B
t1
M1={A,B,t1,Req, MIC[KAB](A,B,t1,Req)}
t2
M2={B,A,t1,t2,t3,Rep,MIC[KAB](B,A,t1,t2,t3,Rep)}
t3
t4
time

Figure 2: (Revised) Secure Pairwise Synchronization (The authors would like to thank Glenn Wurster for pointing out a typo in an earlier version of this figure.)

is not entirely deterministic. When interrupt is disabled, the micro-controller will not be able to get the SFD signal immediately, and the resulting delay will be uncertain.

3.2 Secure Pairwise Synchronization

Given the authenticated MAC layer timestamping capability, we can now describe how two neighbor nodes can perform secure pairwise time synchronization.

Let us take a look at our options. RBS uses a receiver-receiver approach to synchronize nodes [11], in which a reference node broadcasts a reference packet to help pairs of receivers to identify their clock difference. However, an adversary can compromise it by simply launching a pulse-delay attack [12] or wormhole attack against one of the nodes to manipulate the packet transmission delay [19], so that the two nodes receive the reference packet at different times. In some protocols such as FTSP [26], one node passes its own time to the other by directly sending a MAC layer timestamped packet to the latter. This works well in benign environments, as demonstrated in [26]. However, in hostile environments, it suffers from the same problems mentioned above. TPSN uses a sender-receiver approach (through one request and one reply message) to help the sender obtain its clock difference from the receiver [13]. This approach was later improved with security in Secure Pairwise Synchronization (SPS) [12] to deal with pulse-delay and wormhole attacks. Specifically, it authenticates the messages being exchanged, and uses the timestamp information to estimate both the clock difference and the message transmission delay. Pulse-delay and wormhole attacks that manipulate packet transmission delay will introduce extra delay in message transmission, and will be detected.

We adopt the SPS approach [12] with a slight modification. SPS uses a random nonce to prevent replay of a previously transmitted reply message. In our case, we simply use the sender's timestamp in the reply message to prevent replay attacks.

Figure 2 shows the revised SPS protocol, in which all messages are timestamped and authenticated with the key K_{AB} shared by nodes A and B, as described in Section 3.1. Node A initiates the synchronization by sending message M_1. The message contains M_1's sending time t_1. Node B receives the message at t_2. After verifying the message, at time t_3, node B sends a message M_2 that includes t_2, t_3 to node A. (Note that t_1 is also included in M_2 to detect replay attacks.) When node A receives the message at t_4, it can calculate the clock difference $\Delta_{A,B} = \frac{(t_2-t_1)-(t_4-t_3)}{2}$, and the estimated one-way transmission delay $d_A = \frac{t_2-t_1+t_4-t_3}{2}$. Since all messages are authenticated, any modification to any message will be detected. To prevent the pulse-delay attacks [12] and wormhole attacks [19], node A verifies that the one-way transmission delay is less than the maximum expected delay. In fact, this approach can detect any attack that attempts to mislead single-hop pairwise time synchronization by introducing significant extra message delays. As a result, sender A can easily detect attempts to affect the timeliness of the synchronization messages.

Note that the revised SPS protocol only enables the sender (A) to obtain the clock difference with the receiver (B). If the receiver (B) also needs this information, it has to initiate this protocol with the sender (A) as well. An alternative is to perform a three-way message exchange so that both nodes will get the clock difference at the end of the protocol execution. However, in such a three-way protocol, both the sender and the receiver have to maintain their states at the intermediate protocol steps, and each node has to carefully

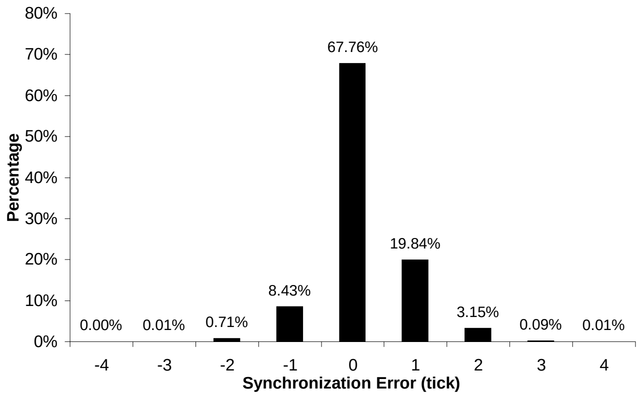


Figure 4: Synchronization Error Distribution in Secure Pairwise Synchronization (1 Tick = 8.68 μs)

maintain its states to avoid interference when it is involved in multiple concurrent synchronizations with different neighbors. The additional space requirement and the increased software complexity do not strictly justify the possibly reduced communication overhead.

Despite the use of MAC layer timestamping, there is still possible uncertainty in pairwise time synchronization, which may affect the precision of time synchronization.

Figure 3 shows the sources of packet delays in the revised SPS protocol. On the target platform (i.e., MICAz motes), the time for the CPU to access current time is deterministic and less than 1 tick. The time for adding timestamp in the buffered packet is deterministic, too, which is less than 2 ticks. The encoding time is for the radio to encode and transform a part of the packet (4 bits a time in case of CC2420) to electromagnetic waves, and the decoding time is for the radio to transform and decode electromagnetic waves into binary data. These times are controlled by a chip sequence at 2 MChips/s, and thus are deterministic. The propagation time depends on the distance between the sender and the receiver, which is also deterministic. The uncertainty in the packet delay is mainly because of the jitter of the interrupt handling, which is caused by temporarily disabled interrupt handling on the receiving side. For example, the receiver may be executing a piece of code that disables interrupt handling when the SFD is received by the RF module, and thus cannot access the receiving timestamp promptly.

We tested 30 pairs of nodes in our lab to obtain the synchronization precision. For each pair of nodes, we ran 500 rounds of pairwise time synchronization. After two nodes finish a pairwise time synchronization, a third reference node broadcasts a query to them. Each of the node records the MAC layer receiving time of the broadcast message and sends the receiving time to the reference node. This allows the reference node to calculate the synchronization error. Figure 4 shows the distribution of the pairwise synchronization error.

4. PHASE II: SECURE AND RESILIENT GLOBAL TIME SYNCHRONIZATION

In this section, we present our method for secure and resilient global time synchronization, assuming all the sensor nodes perform secure single-hop time synchronization periodically.

4.1 Basic Approach

Given the secure pairwise synchronization protocol, the remaining threats to global time synchronization are twofold. First, an external attacker may fake or replay (local) broadcast messages used for global synchronization to mislead the regular nodes. To defend against this threat, we need to integrate authentication of both the *content* and the *timeliness* of broadcast synchronization messages during global time synchronization. Second, a compromised node may provide misleading synchronization information to disrupt the global time synchronization. Thus, our global time synchronization protocol must be resilient to compromised nodes.

We propose a distributed, resilient protocol integrated with (local) broadcast authentication to provide secure and resilient global time synchronization. Intuitively, the source node broadcasts synchronization messages periodically to adjust the clocks of all sensor nodes. The synchronization messages are flooded throughout the network to reach nodes that cannot communicate with the source node directly. Specifically, when receiving a synchronization message for the first time, each node rebroadcasts it (after a random delay to avoid collisions). To reduce the impact of processing delays at intermediate nodes on synchronization precision, our approach focuses on the clock differences without involving the delays directly in the computation. The timely transmission of all these messages are authenticated. Moreover, each node obtains synchronization information from multiple neighbor nodes, so that it can tolerate compromised nodes to a certain extent.

Specifically, each node i maintains a *local clock* C_i. The local clock C_S of the source node S is the desired global clock. For each neighbor node j, each node i can obtain a *single-hop pairwise clock difference* $\Delta_{i,j} = C_j - C_i$ using the secure pairwise time synchronization in Section 3. Each node i maintains a *source clock difference* $\delta_{i,S}$ between its local clock and the clock of the source node S. Node i can directly obtain it if it is a neighbor node of S. Otherwise, node i needs to estimate $\Delta_{i,S}$. When the source node S decides to send a synchronization message, it broadcasts such a message to its direct neighbors. Each of the neighbors can determine its source clock difference directly as mentioned earlier. These neighbor nodes then broadcast their source clock differences to help those that cannot receive the synchronization messages from S directly.

To tolerate up to t compromised neighbor nodes, each node i that receives source clock differences from non-source neighbor nodes computes at least $2t + 1$ *candidate* source clock differences through different neighbor nodes. Specifically, for each source clock difference $\Delta_{j,S}$ that node i receives from node j, node i computes a candidate source clock difference $\Delta^j_{i,S} = C_S - C_i = (C_S - C_j) + (C_j - C_i) = \Delta_{j,S} + \Delta_{i,j}$. Given at least $2t+1$ such candidate clock differences, node i picks the median as the estimated source clock difference $C_{i,S}$. It is easy to see that node i can tolerate up to t compromised neighbor nodes that provide misleading synchronization information.

Each node i can estimate the *global clock* by using its local clock and its source clock difference, i.e., the *estimated global clock* $\hat{C}^i_S = C_i + \Delta_{i,S}$. After computing its own source clock

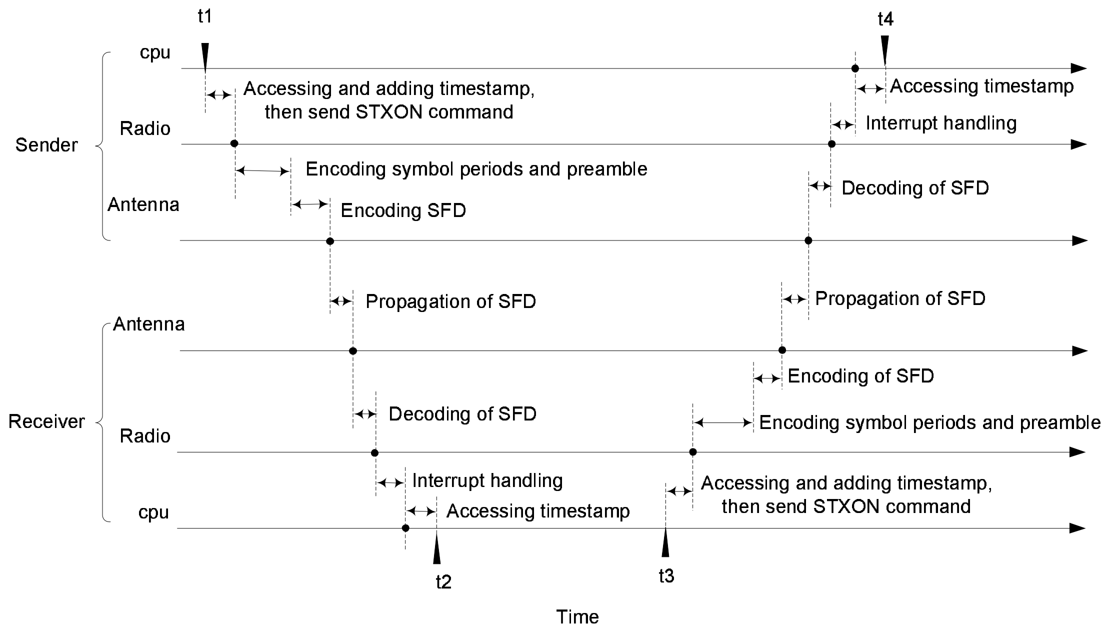


Figure 3: Delay Uncertainty (Certain Details are Omitted for Clarity)

difference, each node broadcasts this value to help the nodes that have not been synchronized estimate their source clock differences as well as the global clock.

Our approach is similar to the resilient clock estimation approach proposed in [41] in that both approaches estimate the clock difference between each node and the source node through multiple nodes that have already been synchronized. However, our approach has a critical difference from that approach: Our approach uses *authenticated local broadcast* to propagate synchronization messages, while the approach in [41] uses authenticated unicast that leads to substantial communication overhead as well as message collisions as shown in the performance study in [41]. This difference represents a key step that enables practical secure and resilient time synchronization in sensor networks.

The ability to authenticate local broadcast messages is the cornerstone of the proposed protocol. In the following, we describe in detail how this is done in TinySeRSync.

4.2 Authentication of Local Broadcast Synchronization Messages

As discussed earlier, the signaling messages for global time synchronization are broadcast in nature, and must be transmitted in a timely and authenticated way. There are two general solutions for authenticating broadcast messages in sensor networks: digital signatures and μTESLA [34, 36]. Though it is possible to verify digital signatures on sensor platforms, as shown in [15], signature operations are still multiple order of magnitude more expensive than secret key based solutions such as μTESLA. Using digital signatures for time synchronization may quickly exhaust the battery power of sensor nodes. Moreover, it is also an attractive target of Denial of Service (DoS) attacks: An attacker may broadcast synchronization messages with false digital signatures to force sensor nodes to perform expensive signature verifications.

μTESLA [36] relies on symmetric cryptography, and thus does not suffer from the above problems. However, μTESLA requires loose time synchronization between the broadcast sender and the receivers. Considering the goal of having the source node synchronize the clocks of all the sensor nodes, there seems to be a conflict in using μTESLA for authenticating broadcast time synchronization messages.

We can indeed avoid the above conflict. We observe that two neighbor nodes may securely perform single-hop pairwise time synchronization using the techniques in Section 3. Consider an arbitrary node A. Assume node A have synchronized with all its neighbor nodes so that node A and any of its neighbor nodes know the clock difference between them. As a result, if node A needs to broadcast a synchronization message to all its neighbor nodes, it may certainly use μTESLA for broadcast authentication, since the "loose synchronization" requirement needed by μTESLA is already satisfied. In other words, we only use μTESLA *locally* to avoid the above conflict.

Specifically, we adapt μTESLA for local broadcast authentication to protect the broadcast messages from a node to its neighbors, assuming the Phase I neighbor synchronization has completed. In the following, we first give a brief introduction to μTESLA, and then discuss the adaptation of μTESLA in TinySeRSync.

4.2.1 Overview of μTESLA

An asymmetric mechanism such as public key cryptography is generally required for broadcast authentication [34]. Otherwise, a malicious receiver can easily forge any message from the sender, as discussed earlier. μTESLA introduces asymmetry by delaying the disclosure of symmetric keys [36]. A sender broadcasts a message with a MIC generated with a secret key K, which is disclosed after a certain period of time. When a receiver gets this message, if it can ensure that the message was sent before the key was disclosed, the receiver buffers this message and authenticates the message when it later receives the disclosed key. To

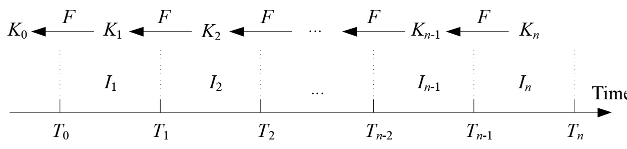


Figure 5: μTESLA

continuously authenticate broadcast messages, μTESLA divides the time period for broadcast into multiple intervals, assigning different keys to different time intervals. All messages broadcast in a particular time interval are authenticated with the key assigned to that time interval.

To authenticate the broadcast messages, a receiver first authenticates the disclosed keys. μTESLA uses a one-way key chain for this purpose. The sender selects a random value K_n as the last key in the key chain and repeatedly performs a (cryptographic) hash function F to compute all the other keys: $K_i = F(K_{i+1}), 0 \leq i \leq n-1$, where the secret key K_i (except for K_0) is assigned to the i-th time interval. Because of the one-way property of the hash function, given K_j in the key chain, anybody can compute all the previous keys $K_i, 0 \leq i \leq j$, but nobody can compute any of the later ones $K_i, j+1 \leq i \leq n$. Thus, with the knowledge of the initial key K_0, which is called the *commitment* of the key chain, a receiver can authenticate any key in the key chain by merely performing hash function operations. When a broadcast message is available in the i-th time interval, the sender generates a MIC for this message with a key derived from K_i, broadcasts this message along with its MIC, and discloses the key K_{i-d} for time interval I_{i-d} in the broadcast message (where d is the disclosure lag of the authentication keys). Figure 5 illustrates the division of the time line and the assignment of authentication keys in μTESLA.

Each key in the key chain will be disclosed after some delay. As a result, the attacker can forge a broadcast message by using the disclosed key. μTESLA uses a security condition to prevent such situations. When a receiver receives an incoming broadcast message in time interval I_i, it checks the security condition $\lfloor (T_c + \Delta - T_1)/T_{int} \rfloor < i + d - 1$, where T_c is the local time when the message is received, T_1 is the start time of the time interval 1, T_{int} is the duration of each time interval, and Δ is the maximum clock difference between the sender and itself. If the security condition is satisfied, i.e., the sender has not disclosed K_i yet, the receiver accepts this message. Otherwise, the receiver simply drops it.

4.2.2 Short Delayed μTESLA: Adapting μTESLA for Global Synchronization

Distribution of μTESLA Parameters: In order to use μTESLA, the sender needs to transmit a number of parameters to all the receivers before the actual broadcast messages. These include the key chain ID, the key chain commitment, the duration of each time interval, and the starting time of the first time interval. We can fix the duration of time intervals and the length of each key chain as network wide parameters. However, the other parameters have to be communicated from each node to its neighbors. To reduce communication cost, we piggy-back the transmission of these μTESLA parameters with the single-hop pairwise synchronization between neighbors. In other words, each node sends the parameters of its own μTESLA key chain to a neighbor node during secure single-hop pairwise synchronization. When one key chain is about to expire, each node needs to communicate with each neighbor node again to transmit the parameters for the next key chain.

Balancing Key Chain Size and Authentication Delay: A direct application of μTESLA to authenticate the local broadcast synchronization messages faces a risk. μTESLA is subject to DoS attacks [35], in which an attacker overhearing a valid broadcast message may use the disclosed key in the message to forge broadcast synchronization messages. A receiver has to buffer all such (forged) messages claimed to be from some neighbor until it receives the disclosed key. As a result, the receiver may not have enough memory to buffer synchronization messages from other neighbor nodes. The immediate authentication mechanism proposed in [35] cannot be applied here, because it requires that the sender know the next message to be transmitted before sending the current message.

One possible way to mitigate the threat of DoS attacks in global synchronization is to exploit the tight time synchronization established during Phase I. Specifically, when using μTESLA for local broadcast authentication, we may use very short time intervals to limit the duration vulnerable to DoS attacks. Because the neighbor nodes have been tightly synchronized with each other during phase I, the broadcast sender can use very short time intervals and disclose an authentication key right after the corresponding interval is over. When the time interval is short enough, it does not give enough time to an attacker to forge broadcast messages using the disclosed key it just learns from the valid broadcast message. A short enough interval duration also offers authentication of the *timeliness* of the synchronization messages; it disallows a replayed message to be transmitted in the valid time interval, and thus enables receivers to detect and remove them.

However, this approach comes with a significant cost: To cover a certain period of time (e.g., 30 minutes), the sender needs to generate a fairly long key chain due to the short time intervals, and most of the keys will be wasted. Reducing the key chain length will force al the neighbor nodes to exchange the key chain commitments frequently, leading to heavy communication overhead.

We propose to adapt the μTESLA broadcast authentication protocol to address the above conflict. Specifically, we propose to use two different durations in one μTESLA instance, a short duration r and a long duration R. The short intervals and the long intervals are interleaved, as shown in Figure 6. As in the original μTESLA, each time interval is still associated with an authentication key, which is used to authenticate messages sent in this time interval. Each node broadcasts a message authenticated with μTESLA only during the short intervals, while broadcasting the disclosed key in the following long interval (possibly multiple times to tolerate message losses).

Upon receiving a broadcast message, a receiver first checks the security condition using the (MAC layer) message receipt time. Because each receiver and the sender have synchronized tightly with each other, the receiver can easily transform the receipt time into the time point in the sender's clock, and verify if the corresponding authentication key has been disclosed when the receiver receives the message.

Consider Figure 6. Suppose the receiver B receives a

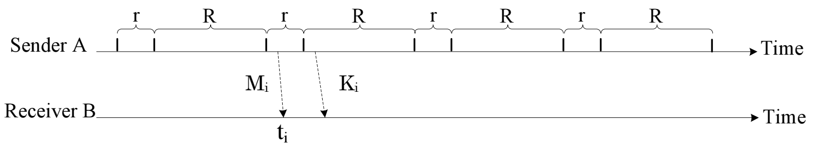


Figure 6: Short Delayed μTESLA

synchronization message M_i from the sender A at its local time t_i (taken in the MAC layer), and the start time of A's μTESLA instance is T_0 in A's clock. B may calculate $i = \lfloor \frac{t_i - T_0}{r+R} \rfloor$ and checks the following security condition: $t_i - T_0 + \Delta_{B,A} + \delta_{max} < i * (R + r) + r$, where $\Delta_{B,A}$ is the pairwise clock difference between A and B, and δ_{max} maximum synchronization error between two neighbor nodes. B stores the message and i only if this check is successful. Otherwise, B simply drops the message. After node B obtains the disclosed key K_i, it verifies $F^{i-j}(K_i) = K_j$ with a previously received key or commitment K_j where $j < i$. If the key is valid, B then uses K_i to verify the MIC included in the broadcast synchronization message M_i.

5. ANALYSIS

5.1 Security Analysis

Phase I. Phase I uses hardware-assisted inline authentication, providing authentication of the source and the content of synchronization messages. Moreover, Phase I uses a two-way message exchange to estimate both the clock difference between direct neighbors and the transmission delay, and can detect attacks that attempt to mislead time synchronization by introducing extra message delays. Thus, Phase I provides protection of the source, the content, and the timeliness of single-hop pairwise synchronization messages. Specifically, Phase I effectively defeats external attacks that attempt to mislead single-hop pairwise time synchronization, including forged and modified messages, pulse-delay attacks, and wormhole attacks that introduce extra delays. Phase I protocol cannot handle DoS attacks that completely jam the communication channel. Nevertheless, no existing protocol can survive such extreme DoS attacks.

Phase II. Phase II adapts μTESLA to provide local broadcast authentication. The security of this μTESLA variation follows directly from the original scheme [34]. Besides local broadcast authentication, another benefit of using μTESLA is the authentication of the timeliness of local broadcast synchronization messages, since a delayed message will be automatically discarded due to the violation of the security condition. Thus, similar to Phase I, by authenticating the source, the content, and the timeliness of local broadcast synchronization messages, Phase II can successfully defeat all the external attacks that are intended to mislead the time synchronization.

Since the source node is trusted, in Phase II, each direct neighbor node of the source node can directly estimate the global clock securely. However, the other nodes may receive false synchronization information from compromised nodes. The solution used by Phase II is to have each node use the source clock differences received from $2t + 1$ neighbor nodes to estimate $2t + 1$ candidate source clock differences, and select the median one as its own source clock difference.

We say a source clock difference obtained by a normal node is *correct* if it has no more error than one obtained using information *only* from normal nodes. The resilience property of the Phase II protocol can be seen by induction. As discussed earlier, the source node is trusted. Consider a normal node i. Suppose the source clock difference is obtained through node j, that is, $\delta_{i,S} = \delta_{i,j} + \delta_{j,S}$. There are two cases. (1) If node j is a normal node, both $\delta_{j,S}$ and $\delta_{i,j}$ must be correct according to the induction assumption, and thus $\delta_{i,S} = \delta_{i,j} + \delta_{j,S}$ is correct by definition. (2) Suppose node j is malicious. Because there are at most t malicious nodes, $\delta_{i,S}$, which is the median of the $2t+1$ candidate source clock differences, must be between two candidate source clock differences obtained through two normal nodes. This implies that the synchronization error introduced by a compromised node is no more than the error introduced by a normal node. As a result, the source clock difference $\delta_{i,S}$ is still correct. Thus, if every node has no more than t compromised neighbor nodes, Phase II can successfully synchronize all the normal nodes as long as they have enough number of neighbor nodes. However, similar to Phase I, Phase II cannot handle DoS attacks that completely jam the communication channel.

In conclusion, TinySeRSync provides a comprehensive solution to providing secure and resilient time synchronization in wireless sensor networks. It can successfully defeat all non-DoS external attacks against time synchronization, and is resilient to compromised nodes.

5.2 Performance Analysis

Synchronization Precision and Coverage. TinySeRSync uses predication-based MAC layer timestamping in Phase I, avoiding many places that could introduce uncertainty during time synchronization. In Phase II, TinySeRSync tries to estimate the global clock through the estimation of source clock differences, and thus greatly reduces the impact generated by the propagation delays of synchronization messages. Thus, TinySeRSync can provide high precision time synchronization. Moreover, TinySeRSync employs flooding-based propagation of global synchronization messages; this allows all the nodes that have enough number of neighbor nodes to be synchronized.

Communication, Computation, and Storage Overheads. TinySeRSync uses message exchanges between direct neighbor nodes for Phase I synchronization. All these message exchanges are local, and do not introduce wide area interference. In Phase II, TinySeRSync adopts local broadcast for the propagation of global synchronization messages, effectively harnessing the broadcast nature of wireless communication. Thus, TinySeRSync is efficient in terms of communication.

TinySeRSync uses efficient symmetric cryptography for message authentication. In particular, it exploits the hardware cryptographic support provided by the CC2420 radio component. Thus, TinySeRSync introduces very light computation overhead for cryptographic operations.

TinySeRSync does increase the storage overhead on sensor nodes due to the need to maintain cryptographic keys, buffer the local broadcast messages, and store the source clock differences received from $2t + 1$ neighbor nodes. A critical issue is the maintenance of the μTESLA key chain required for authenticating outgoing synchronization messages. Our adaptation of μTESLA greatly reduces the number of keys in each key chain. In addition, we use another approach to further reduce the memory requirement and the delay: After generating a key chain, each node only saves some select keys called *key anchors* (e.g., 1 of every 10 keys), and also caches the keys before the next key anchor to be used (e.g., the first 10 keys). When a μTESLA key is required for authentication, if the key is available in the cache, the node can directly use it. Otherwise, the node can regenerate and fill the key cache using the next key anchor.

Incremental Deployment. As discussed earlier, TinySeRSync uses two asynchronous phases, both of which are executed periodically. Thus, TinySeRSync works well with incremental deployment of sensor nodes. The newly deployed nodes first obtain the pairwise time differences and the commitments of the key chains from its neighbor nodes in Phase I, and then join the Phase II global time synchronization. Our experimental results in Section 7 will show the performance when there are incrementally deployed nodes.

6. IMPLEMENTATION DETAILS

Our implementation of TinySeRSync is targeted at MICAz motes [2]. (However, our implementation can be used with slight modification for other sensor platforms that also use CC2420 radio components, such as TelosB [4] and Tmote Sky [5].) A MICAz mote has an 8-bit micro-controller *ATMega128L* [1], which has 128 kB program memory and 4 kB SRAM. As discussed earlier, MICAz is equipped with the ChipCon CC2420 radio component [3], which works at 2.4GHz radio frequency and provides up to 250 kbps data rate. CC2420 is an IEEE 802.15.4 compliant RF transceiver that features hardware security support.

In the following, we give a few details that are critical for repeating our implementation.

6.1 Exploiting Hardware Security Support in CC2420

The hardware security support featured by CC2420 provides two types of security operations: *stand-alone encryption operation* and *in-line security operation*. The stand-alone encryption operation provides a plain AES encryption, with 128 bit plaintext and 128 bit keys. To encrypt a plaintext, a node first writes the plaintext to the stand-alone buffer *SABUF*, and then issues a SAES command to initiate the encryption operation. When the encryption is complete, the ciphertext is written back to the stand-alone buffer, overwriting the plaintext.

The in-line security operations can provide encryption, decryption, and authentication on frames within the receive buffer (RXFIFO) and the transmit buffer (TXFIFO) of CC2420 on a per frame basis. It supports three modes of security: *counter mode (CTR)*, *CBC-MIC*, and *CCM*. CTR mode performs encryption on the outgoing MAC frames in the TXFIFO buffer, and performs decryption on the incoming MAC frames in the RXFIFO buffer. CBC-MIC mode can generate and verify the message integrity code (MIC) of the messages. The length of MIC can be adjusted. CCM mode combines CTR mode encryption and CBC-MIC authentication in one operation. All the three security modes are based on AES encryption/decryption using 128 bit keys.

We use the CBC-MIC mode to authenticate both pairwise and global synchronization messages. A sender can use in-line CBC-MIC mode to generate the MIC for both pair-wise and global synchronization messages in the MAC layer after the message has been written to the TXFIFO buffer.

The receiver side, however, is slightly different. When a receiver receives a pair-wise synchronization message, since it already knows the secrete key shared with the sender, it can use the in-line CBC-MIC mode to verify the MIC before the message is read from the RXFIFO buffer. However, for the global synchronization messages, before receiving the disclosed key, the receiver cannot use in-line authentication to verify the MIC in the message. Because the receiver still needs the RXFIFO buffer to receive other messages, it cannot buffer the message in the RXFIFO buffer while waiting for the disclosed key. Thus, we have the receiver read the message from RXFIFO and buffer it in its local memory. When the key is received, the receive uses the stand-alone mode to authenticate the buffered global synchronization messages. Since the stand-alone mode only provides single-block encryption functionality, we implemented the CBC mode based on the hardware support.

6.2 Handling Timers

Using timers on MICAz is a tricky issue; improper uses usually lead to unexpected results. The micro-controller ATMega128 provides two 8-bit timers (Timer0, Timer2) and two 16-bit timers (Timer1, Timer3) [1]. In TinyOS, Timer 0 is mainly used as one-shot or repeat timers for applications. For MICAz, Timer 2 is used by CC2420 as a high precision timer (32 μs per tick) to back off the sending packets for a short period of time. Timer 1 is used by CC2420 for capturing radio packet transmit and receive events. In TinySeRSync, we use the remaining 16-bit Timer 3 to maintain the local clock and schedule the message transmission.

ATMega128L uses a 7.3728 MHz crystal oscillator as I/O clock source, whose accuracy is $\pm 40 ppm$ [3]. In our implementation, we divide the I/O clock by 64 as the source of Timer 3, thereby achieving a 115.2 kHz Timer 3, with a 8.68 μs time resolution. Timer3 provides three compare match registers (*OCR3A/B/C*), each connected with an interrupt vector. If the compare match interrupt is enabled, whenever the value of Timer3 (TCNT3) equals to the value of one compare match register, it will trigger an interrupt to handle the event. Each node uses compare match register A to maintain a 48-bit logical clock. The value of Timer3 (TCNT3) is 16 bits, and it will overflow every 568.8 ms. We add another 32 bits to have a logical clock that will not overflow for over 77 years Each node sets compare match register B to launch pair-wise synchronization with its neighbors periodically. The source node will use compare match register B to initiate the global synchronization periodically. Each node uses compare match register C to send its global synchronization message in its nearest short μTESLA interval and disclose the key in the adjacent long μTESLA interval.

7. EXPERIMENT RESULTS

We performed a series of experiments in a network of 60 MICAz motes to evaluate the performance of TinySeRSync in real deployment. We focused on the performance met-

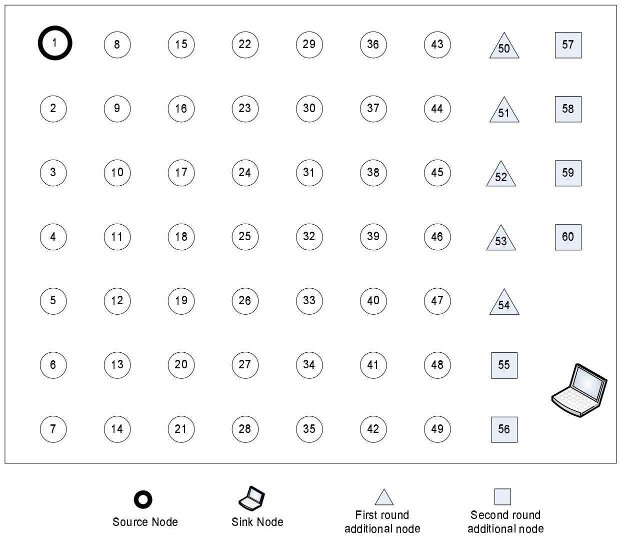


Figure 7: Network Topology

Table 1: Code Size

Memory	Size (bytes)
RAM	1,977
ROM	24,814

rics in normal situations, while relying on the analysis in Section 5 for the security properties.

7.1 Configuration

Figure 7 shows the sensor network test-bed used in our experiments. (The different node shapes represent nodes deployed at different times during incremental deployment, which will be explained in Section 7.4.) The test-bed consists of 60 nodes, among which node 1 (with the solid circle) is configured as the source node.

We use a number of parameters in our evaluation. Each node performs a secure single-hop pairwise synchronization with its neighbor nodes for every $d_1 = 4$ seconds. During this synchronization, the node informs its neighbor nodes its μTESLA parameters. The source node starts a global synchronization every d_2 seconds. In our experiments, we use $d_2 = 5$ or 10 seconds. The degree of tolerance (against compromised neighbor nodes) is represented as t, as used throughout this paper. In our experiments, we use $t = 0, 1, 2, 3, 4$ to examine the various performance metrics.

We use a sink node to help collecting data from each sensor node. Periodically, the sink node broadcasts an anchor message with the highest power to all the nodes. Upon receiving this message, each node marks the receiving time and converts it to the global time using its source clock difference. The sink node then queries each node individually to get the receiving time (in the estimated global clock) along with other auxiliary information. This allows us to discover the synchronization error on each individual sensor node, the synchronization coverage, as well as the number of synchronization levels each node has to go through.

7.2 Code Size

Let us first look at the code size before presenting the performance results. The code size is related to the maximum number of compromised nodes we would like to tolerate. For each neighbor node, a node will spend 46 bytes to save the pairwise key, current key in key chain, and clock differences, etc. In our experiments, each node saves 10 keys for a key chain with 100 keys. Each node reserves a buffer to store at most 6 unauthenticated global synchronization messages, which increase the size of RAM.

7.3 Performance in Static Deployment

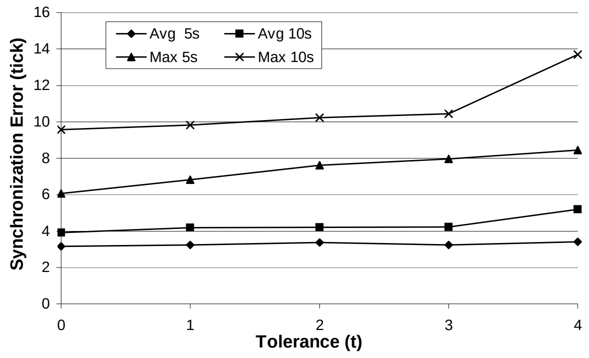
Let us first look at the performance of TinySeRSync in static deployments. In our experiments, we use the following metrics to evaluate the performance and the overhead of TinySeRSync: the average and the maximum synchronization errors, the synchronization rate (i.e., the percentage of nodes that can be synchronized), the synchronization level (i.e., the maximum number of hops that global synchronization messages have to go through before a sensor node can be synchronized.

Average and Maximum Synchronization Error: Figure 8(a) shows the maximum and the average synchronization error with different global synchronization intervals and different degrees of tolerance against compromised neighbor nodes. In all cases, the maximum synchronization error is below 14 ticks (121.52 μs), and the average synchronization error is below 6 ticks (52.08 μs). Figure 8(a) also shows that as the global synchronization interval increases, the maximum and the average synchronization errors both increase.

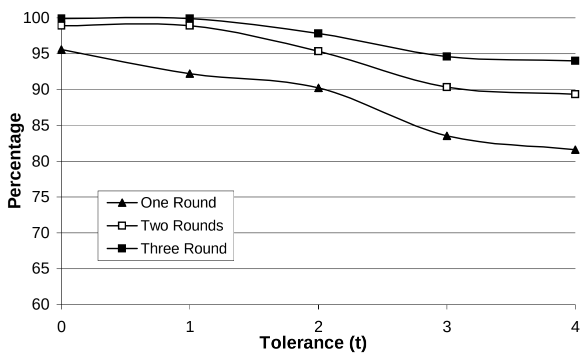
Synchronization Rate: Figure 8(b) shows the synchronization rate (i.e., the percentage of nodes that can be synchronized by TinySeRSync) after one, two, and three rounds of global synchronization. When the tolerance against compromised neighbor nodes increases, as we expected, the synchronization rate decreases. However, after three rounds of global synchronization, even in the worst case, about 95% of the nodes can be synchronized to the source node.

Synchronization Level: Figure 9(a) shows the maximum and the average number of hops the global synchronization messages have to traverse before all the nodes are synchronized. In our test-bed, in all cases, the average synchronization level is around 3. An interesting issue is that the maximum synchronization level initially decreases as the tolerance t increases, but then goes up as t is greater than 2. This is because when t is very small (i.e., $t = 0, 1$), a node can broadcast the synchronization message almost immediately after it is synchronized. The synchronization triggered by these fast nodes may be propagated to many nodes that have not been synchronized. However, when t is large enough, synchronizing a node with increased t requires receiving synchronization messages from more neighbor nodes, thus resulting in an increasing trend for maximum synchronization levels.

Communication Overhead: We measure the communication overhead by assessing the number of messages each node has to transmit per time unit. For each neighbor node, a node sends one message to obtain the pairwise time difference. In one round of global time synchronization, each node at most broadcasts one synchronization message and one key disclosure message. Suppose each node has n neighbor nodes, the pairwise synchronization interval is d_1, and the global synchronization interval is d_2. In a given long time interval T, each node sends at most $n \cdot \frac{T}{d_1} + \frac{2T}{d_2}$ messages. Figure 9(b) shows the communication overhead per hour for

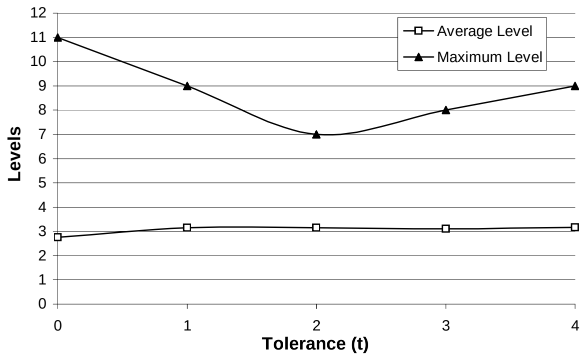


(a) Maximum and average synchronization error

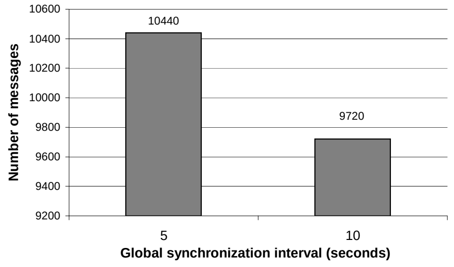


(b) Synchronization rate

Figure 8: Synchronization Error and Synchronization Rate



(a) Maximum and average synchronization level



(b) Communication overhead (# messages each node sends per hour)

Figure 9: Synchronization Level and Communication Overhead

a configuration where each node has 10 neighbor nodes, the pairwise time synchronization interval is 4 seconds, and the global time synchronization interval is 10 seconds.

7.4 Incremental Deployment

We evaluated the performance of TinySeRSync when there were incremental deployments. Consider Figure 7. At the beginning of the experiment, we deployed the 49 nodes marked as circles. We then added 5 new nodes into the network about 10 minutes later, and added another 6 new nodes about 1 minute later. In this experiment, we set $t = 2$, and the global synchronization interval is set to 10s. Figure 10 shows the history of the average synchronization error and the coverage in this experiment. As shown in the figure, when new nodes were just added into the network, they could not be synchronized immediately, and the average synchronization error was large and the synchronization rate dropped to around 90%. However, after a few rounds of global synchronization, all these new nodes were correctly synchronized, resulting in a low average synchronization error and 100% synchronization coverage.

8. CONCLUSION AND FUTURE WORK

In this paper, we presented the design, implementation, and evaluation of TinySeRSync, a secure and resilient time synchronization subsystem for wireless sensor networks running TinyOS. TinySeRSync includes a comprehensive suite of techniques, including a secure single-hop pairwise time synchronization protocol based on hardware-assisted, authenticated MAC layer timestamping, and a secure and resilient global time synchronization protocol based on a novel use of the μTESLA broadcast authentication protocol. These techniques exceed the capability of previous solutions. In particular, unlike the previous attempts, the secure single-hop pairwise synchronization technique can handle high data rate such as those produced by MICAz motes (in contrast to those by MICA2 motes). Moreover, our novel use of μTESLA in global time synchronization successfully resolved the conflict between the goal of achieving time synchronization and the fact that μTESLA requires loose time synchronization. The resulting protocol is secure against external attacks and resilient against compromised nodes.

Our future research is two-fold. First, we will investigate additional techniques that can improve the synchronization precision. One potential solution is to adapt the linear regression technique proposed in [26] to compensate the constant clock drifts. Second, we will look into the integration of TinySeRSync in sensor network applications.

9. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments.

10. REFERENCES

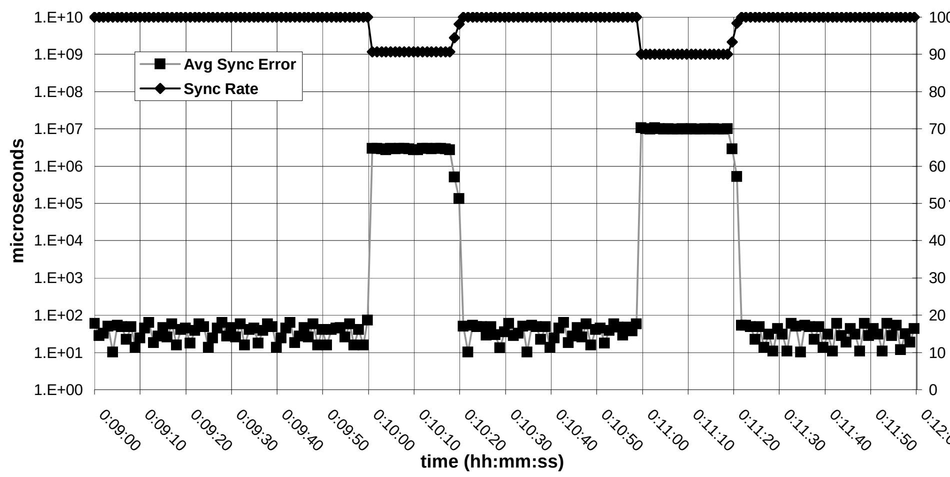


Figure 10: Average Synchronization Error (Left Y-axis) and Coverage (Right Y-axis) During Incremental Deployment ($t = 2$)

[1] ATmega128(L) Complete Technical Documents. `http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf`.

[2] MICAz: Wireless measurement system. `http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf`.

[3] SmartRF CC2420 Datasheet (rev 1.3), 2005-10-03. `http://www.chipcon.com/files/CC2420_Data_Sheet_1_3.pdf`.

[4] TelosB mote platform. `http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/TelosB_Datasheet.pdf`.

[5] Tmote Sky: Reliable low-power wireless sensor networking eases development and deployment. `http://www.moteiv.com/products-tmotesky.php`.

[6] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: A survey. *Computer Networks*, 38(4):393–422, 2002.

[7] B. Barak, S. Halevi, A. Herzberg, and D. Naor. Clock synchronization with faults and recoveries. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 133–142, 2000.

[8] Crossbow Technology Inc. Wireless sensor networks. `http://www.xbow.com/Products/Wireless_Sensor_Networks.htm`.

[9] D. Dolev, J. Y. Halpern, B. Simons, and R. Strong. Dynamic fault-tolerant clock synchronization. *Journal of the ACM*, 42(1):143–185, 1995.

[10] J. R. Douceur. The sybil attack. In *First International Workshop on Peer-to-Peer Systems (IPTPS'02)*, Mar 2002.

[11] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. *ACM SIGOPS Operating Systems Review*, 36:147–163, 2002.

[12] S. Ganeriwal, S. Capkun, C. Han, and M. B. Srivastava. Secure time synchronization service for sensor networks. In *Proceedings of 2005 ACM Workshop on Wireless Security (WiSe 2005)*, pages 97–106, September 2005.

[13] S. Ganeriwal, R. Kumar, and M. B. Srivastava. Timing-sync protocol for sensor networks. In *Proceedings of the First International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 138–149, 2003.

[14] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of Programming Language Design and Implementation (PLDI 2003)*, June 2003.

[15] N. Gura, A. Patel, A. Wander, H. Eberle, and S.C. Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES 2004)*, August 2004.

[16] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D.E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[17] A. Hu and S. D. Servetto. Asymptotically optimal time synchronization in dense sensor networks. In *Proceedings of the Second ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, pages 1–10, September 2003.

[18] Y. Hu, A. Perrig, and D. B. Johnson. Wormhole detection in wireless ad hoc networks. Technical Report TR01-384, Department of Computer Science, Rice University, Dec 2001.

[19] Y.C. Hu, A. Perrig, and D.B. Johnson. Packet leashes: A defense against wormhole attacks in wireless ad hoc networks. In *Proceedings of INFOCOM 2003*, April 2003.

[20] IEEE Computer Society. IEEE 802.15.4: Ieee standard for information technology – telecommunications and information exchange between systems local and metropolitan area networks – specific requirements part 15.4: Wireless medium access control (MAC) and physical layer (PHY) specifications for low-rate wireless personal area networks (LR-WPANs).

http://standards.ieee.org/getieee802/download/802.15.4-2003.pdf, October 2003.

[21] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of the sixth annual international conference on Mobile computing and networking (Mobicom '00)*, pages 56–67, August 2000.

[22] L. Lamport and P.M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, 1985.

[23] Q. Li and D. Rus. Global clock synchronization in sensor networks. In *Proceedings of IEEE INFOCOM 2004*, pages 214–226, March 2004.

[24] D. Liu, P. Ning, and R. Li. TinyKeyMan: Key management for sensor networks. http://discovery.csc.ncsu.edu/software/TinyKeyMan/.

[25] M. Manzo, T. Roosta, and S. Sastry. Time synchronization attacks in sensor networks. In *Proceedings of the 3rd ACM workshop on Security of ad hoc and sensor networks*, pages 107–116, 2005.

[26] M. Maroti, B. Kusy, G. Simon, and A. Ledeczi. The flooding time synchronization protocol. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*, pages 39–49, Nov 2004.

[27] D.L. Mills. Internet time synchronization: The network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, 1991.

[28] M. Mock, R. Frings, E. Nett, and S. Trikaliotis. Clock synchronization for wireless local area networks. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems (Euromicro-RTS 2000)*, June 2000.

[29] J. Newsome, R. Shi, D. Song, and A. Perrig. The sybil attack in sensor networks: Analysis and defenses. In *Proceedings of IEEE International Conference on Information Processing in Sensor Networks (IPSN 2004)*, April 2004.

[30] J. Newsome and D. Song. GEM: graph embedding for routing and data-centric storage in sensor networks without geographic information. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys '03)*, pages 76–88, Nov 2003.

[31] D. Niculescu and B. Nath. Ad hoc positioning system (APS). In *Proceedings of IEEE GLOBECOM '01*, 2001.

[32] A. Olson and K.G. Shin. Fault-tolerant clock synchronization in large multicomputer systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):912–923, 1994.

[33] S. PalChaudhuri, A.K. Saha, and D.B. Johnson. Adaptive clock synchronization in sensor networks. In *Information Processing in Sensor Networks (IPSN)*, pages 340–348, April 2004.

[34] A. Perrig, R. Canetti, D. Song, and D. Tygar. Efficient authentication and signing of multicast streams over lossy channels. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, May 2000.

[35] A. Perrig, R. Canetti, D. Song, and D. Tygar. Efficient and secure source authentication for multicast. In *Proceedings of Network and Distributed System Security Symposium*, February 2001.

[36] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and D. Tygar. SPINS: Security protocols for sensor networks. In *Proceedings of Seventh Annual International Conference on Mobile Computing and Networks*, pages 521–534, July 2001.

[37] M.L. Sichitiu and C. Veerarittiphan. Simple, accurate time synchronization for wireless sensor networks. In *IEEE Wireless Communications and Networking Conference WCNC03*, 2003.

[38] H. Song, S. Zhu, and G. Cao. Attack-resilient time synchronization for wireless sensor networks. In *Proceedings of IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS'05)*, 2005.

[39] T. K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, 1987.

[40] K. Sun, P. Ning, and C. Wang. Fault-tolerant cluster-wise clock synchronization for wireless sensor networks. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2(3):177–189, July–September 2005.

[41] K. Sun, P. Ning, and C. Wang. Secure and resilient clock synchronization in wireless sensor networks. *IEEE Journal on Selected Areas in Communications*, 24(2), February 2006.