

An Efficient Key Update Scheme for Wireless Sensor Networks

Kamini Prajapati
School of EECS
Washington State University
22200 NE, 9th Drive
Sammamish, WA- 98074

Jabulani Nyathi
School of EECS
Washington State University
P. O. Box 642752
Pullman, WA 99164-2752

Abstract - *Wireless sensors are highly resource constrained in terms of memory, power and processing capability. However, critical applications of wireless sensor networks demand security features to be implemented. Further, when the thrust is miniaturization, provision of security features becomes challenging. Researchers have approached this problem and provided schemes such as SPINS, TinySec, TinyPK, Localized Encryption and Authentication Protocol(LEAP), Elliptical curve Cryptography(ECC), etc. TinySec is the most successful implementation to date. In this paper, we present some enhancements to the basic TinySec security features by providing an efficient key update mechanism on top of TinySec. The simulation results show that the memory overhead for this scheme is 1.66% and the computational cost is minimal. There is no latency or bandwidth overhead. Additionally to being feasible for implementation on current sensors, this scheme will be feasible on future miniaturized sensors with limited resources.*

Keywords: Wireless sensor networks, Key update, TinySec, Sensor network security, Resource consumption.

1 Introduction

Wireless sensor networks or sensor networks as they are usually called, refers to a heterogeneous system consisting of tiny wireless sensors and actuators with general purpose computing elements, [1]. These sensor nodes consist of sensing, data processing, and communication components. Typical application involves deploying hundreds or thousands of low-power, low-cost sensor nodes for a specified purpose, like habitat monitoring, burglar alarms, prognostic health management, battlefield management, etc. For the majority of applications, sensor networks are designed to be unattended for long periods after deployment and battery recharging or replacement may not be possible, [1].

Sensor networks can facilitate large-scale, real-time data processing in complex environments. They interact closely with their physical environment and with people. Their applications involve protecting and monitoring critical military, environmental, safety-critical or domestic infrastructures and resources. Security is important in sensor networks for the following reasons, [2]:

- Since sensor networks actively monitor their surroundings; it is often easy to deduce information other than the data being monitored. Such unwanted information leakage often results in privacy breaches of the people in the environment.
- Wireless communication employed by the sensor networks facilitates eavesdropping and packet injection by an adversary.

The combination of these factors demand security for sensor networks to ensure operation safety, secrecy of sensitive data, and privacy for people in sensor environment. Without proper security mechanisms, sensor networks will be confined to limited controlled environment, thereby restricting their application scope. Therefore, in order to monitor and protect safety-critical resources and structures, security is needed in wireless sensor networks.

Given the importance of security in sensor networks, a good amount of research has been done in providing various security features. However, the resource constrained nature of sensors poses a challenge to provide adequate security with little resource consumption. SPINS, [3] was the first approach to security in sensor networks. Unfortunately, it was never successfully implemented. TinySec, [4], a link layer security mechanism, is the first fully implemented security architecture on sensor networks. Localized Encryption and Authentication Protocol (LEAP), [5], is robust but complex security protocol consuming a considerable amount of resources. Further, there

has been research on key distribution and key update. Camtepe et. al. [6] gives a comprehensive summary of different schemes.

Currently, the most popular *mote* (sensor node) is Mica2, [7]. It is a several cubic inch sensor/actuator unit with a CPU, radio, power supply and optional sensing elements. The processor used is 8-bit 8 MHz Atmel ATMEGA128L CPU with 4 KB of RAM, 128 KB of program memory, and 512 KB of flash memory. The software used on Mica2 is TinyOS, [7], a small, event-driven, component based operating system, designed specifically for networked sensors requiring minimum hardware. At full power, Mica2 can run only for about 2 weeks. We agree with Sastry et. al. [4] that Moore’s law will be used for miniaturization and low cost development of the sensors rather than increase in their performance capabilities. Thus, for successful implementation on sensor networks, the security feature should consume minimal resources.

In this paper, we propose a key update scheme that is based on TinySec. TinySec addresses the basic security needs of message authentication, integrity and confidentiality, [4]. It was designed with the goals of achieving the basic security without causing excessive overhead for the resource constrained sensor networks. Further, being a link layer protocol, it is transparent to all TinyOS applications, and thus, has the scope of widespread deployment. TinySec provides no key update and it is this aspect of sensor network security that we address. TinySec does not guard against resource consumption attacks, physical temper or node capture. Our scheme enhances some of TinySec’s capabilities and to some extent offers resilience against node capture. We stay true to TinySec’s ability to place less strain on a node’s processor, memory and power dissipation.

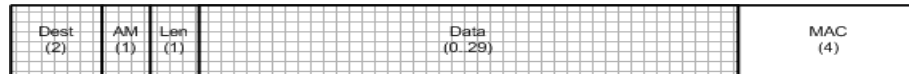
2 Background Work

As mentioned in section 1, our key update scheme is based on TinySec. TinySec provides message integrity and authentication by means of Message authentication Code (MAC), and confidentiality by encryption of data. The default TinyOS radio stack is modified to incorporate TinySec. Current implementation of TinySec uses a 16 byte network-wide shared key which is pre-deployed in the nodes.

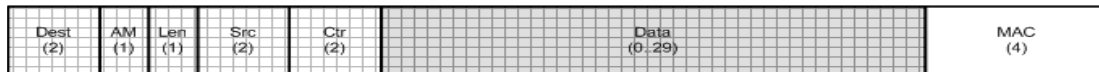
The format for default TinyOS packet is shown in figure 1(a). The destination field is 2 bytes long, the AM, length and group fields are 1 byte each, followed by data payload which could be a maximum up to 29 bytes. TinyOS uses



(a) Default TinyOS packet format



(b) TinySec-Auth packet format



(c) TinySec-AE packet format

Figure 1: The TinyOS and TinySec packet formats

CRC as default mechanism to detect transmission errors. The packet has 2 bytes for CRC. TinySec, [4] replaces CRC with 4 byte MAC to guarantee message integrity and authentication.

TinySec provides two modes for security; TinySec Authentication (TinySec-Auth) and TinySec Authentication and Encryption (TinySec – AE). Figure 1(b) and (c) shows the different fields of each packet format. In both the modes, the MAC is computed over the data payload, however data is encrypted in the AE mode prior to MAC computation. Additionally, the TinySec – AE packet is 4 byte longer than the TinySec - Auth packet. These 4 bytes comprise the Initialization Vector (IV) for SkipJack encryption in Cipher Block Chaining (CBC) mode. Since, the default maximum data payload is 29 bytes, only the last five bits of the 1 byte length field are used. TinySec uses

the first two bits of length field to encode its two modes. We use the remaining unused bit of the length field for our scheme.

3 Proposed Scheme

We employ a simple yet effective key update scheme that works as follows: Whenever the time for key update comes, the base station or the node wishing to do a key update first sets the third bit of the length field. The current encryption key is then rotated. The data is encrypted with this new key and the packet is sent. The receiving nodes decode the length byte first as is done for TinySec without key update. The decoded length byte provides information about TinySec's mode of operation (authentication or authentication with encryption). The mode is determined by examining the first two bits. The status of the third bit indicates whether the key has to be modified or not. Note that the toggling of the third bit of the length field would only occur if the mode of operation is TinySec authentication and encryption (TinySec-AE). If the bit of interest is set, the receiver will first rotate its encryption key, and then decrypt the data. In this way, we achieve two things at the same time; the synchronization and key update. Figure 2(a) and (b) represents the steps involved in key update at the sender and receiver side.

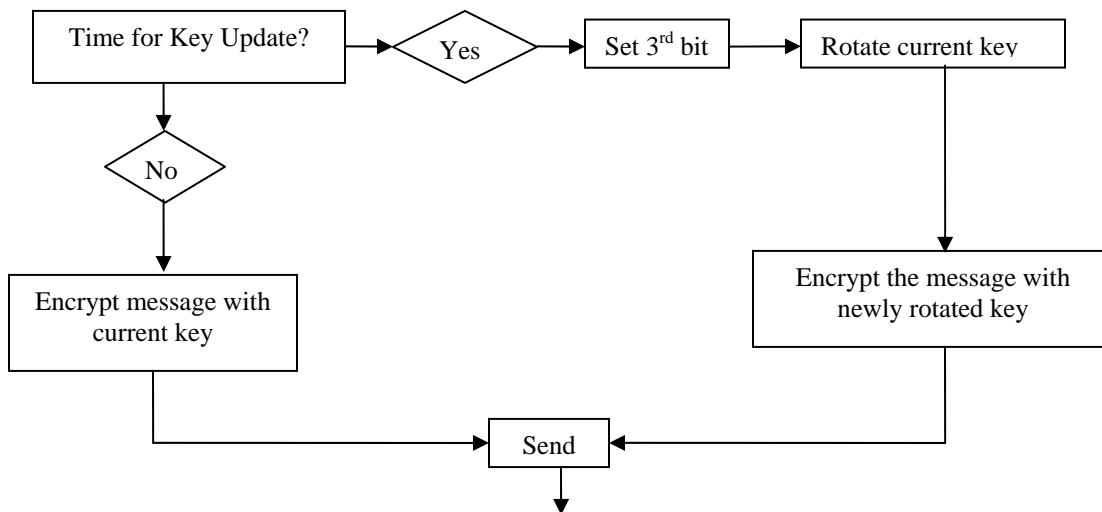


Figure 2(a): Activity on Sender Side

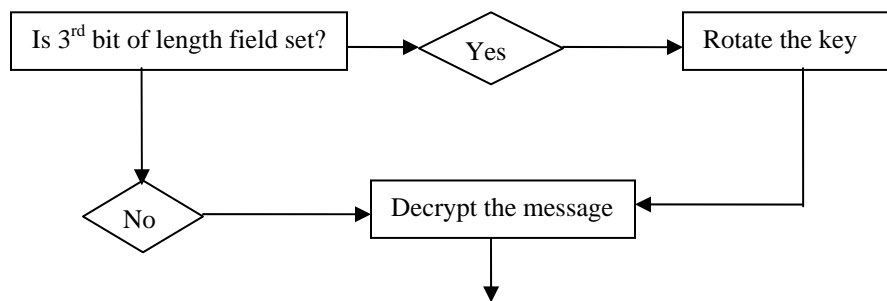


Figure 2(b): Activity at Receiver side

3.1 Cost Analysis

The proposed scheme is very efficient and simple. The key update process does not require extensive computations to generate a new key. Also, the sender need not send any key update message prior to key update. This function is performed by setting the third bit of the length field at the time of key update. Further, no key is sent in the message since the key update is achieved by rotation of bits of the already existing key. In this way, the overhead of sending the key update message and the key are avoided. There is no bandwidth overhead at all. The latency overhead is also avoided since the packet size remains the same as that of TinySec and so no additional time

is required to send the key update request or the key. The operations required for new key computation and energy dissipation involve only:

- Setting the third bit of the length field and
- Performing a bit rotation

These operations require very few instructions. Thus, the computational time is negligible and the power required is extremely minimal. It therefore follows that bandwidth, latency and computational overheads per node are very minimal. The simulation results show that the instructions executed to perform key update require a total of 155 CPU cycles compared to 103756 cycles required to perform encryption as reported in [8]. The percentage increase in memory as compared to TinySec is 1.66% for ROM and 0.347% for RAM.

3.2 Security Analysis

Since we follow the TinySec model, our security provisions are at the same level as those of TinySec with the exception being the capability to update the key. Key updating enhances the confidentiality by minimizing the potential for IV re-use over an extended operation time. There is high probability of IV being reused in TinySec since its counter field is only 2 bytes long and would repeat after 2^{16} packets for a sender. If the IV is reused with the same key, the ciphertexts produced for two identical plaintexts would be the same. This would result in undesirable information leakage.

As in TinySec, we do not address resource consumption attacks and replay attacks. Our scheme addresses node capture attack indirectly; since the time for key update is known only to the sender, if key update is frequent a captured node will have the key decrypted but the adversary might be in possession of an outdated key by the time it attempts to join the network.

In TinySec, a 16 byte key is preloaded into the nodes. The first 8 bytes are used as encryption key and the next 8 bytes are used as the message authentication code (MAC) key. These bytes are copied into their respective buffers. Since there are only 8 bytes for encryption, it follows that there is a limitation on the number of different/unique keys possible. The number of different keys possible using key rotation is 64. We consider this number good enough for sensor networks. The best case occurs if key update is done sparingly for example once per day. In this case it would be 64 days before key re-use. Frequent key update would see the key being re-used sooner however this limitation can be overcome by swapping any two neighboring/intermediate bits once the key update count has reached 64 and then again rotate the key for 64 times.

One aspect of key update is the maintenance of key quality. For any security mechanism, the value of the key plays an important role in providing security strength. Key size and randomness of the key value are the major factors influencing the key quality. A key length of 80 bits is generally considered the minimum for strong security with symmetric encryption algorithms. TinySec takes the 64 bit value and expands it to 80 bits before performing encryption as required in SkipJack Block Cipher. Regarding the key value, according to the SkipJack Review [9], there exist no key value which can be considered as weak key for SkipJack cipher. Even the key with all '0's or all '1's is not weak because of the design of SkipJack algorithm. There is no pattern of symmetry in the SkipJack algorithm which could lead to weak keys.

In the proposed scheme the key is rotated circularly. Depending on the initial value of the preloaded key, the updated key will have various combinations of '1's and '0's. As per [9], any key value will not reduce the strength of encryption/decryption. Further, the number of '1's and '0's present in the original key and the updated key remains same due to rotation. We can also consider other simple alternatives to rotation, for example, incrementing the key value by one for every key update. However, depending on the initial key value, at one point the key value will be all 1's. This would require the initial value of the key to be stored somewhere to prevent the overflow. In spite of this, we can consider increment operation as an option since SkipJack cannot have weak keys. Similarly, we can consider any other simple arithmetic operation for updating the key as far as the key strength is concerned.

We have proposed an efficient and resource aware key update scheme which consumes minimal computational resources, has no additional bandwidth requirements and requires very little additional storage. This scheme is developed with TinySec as the reference security protocol and enhances the confidentiality provided by the protocol.

4 The Simulation Environment

To test our scheme we used TOSSIM—the simulator for TinyOS, [10]. TOSSIM is specifically designed for TinyOS sensor networks and is the de-facto for testing, debugging, and analyzing TinyOS applications. Applications

developed under this environment are to scale i.e. they can be readily downloaded onto the mote from the development system.

TOSSIM, [10] is a discrete-event simulator for TinyOS sensor networks and provides a scalable simulation environment for applications based on TinyOS. It is. Unlike machine-level simulators, TOSSIM compiles a TinyOS application into a native executable that runs on the simulation host. This design allows TOSSIM to be extremely scalable, supporting thousands of simulated nodes. It compiles directly from TinyOS code. Deriving the simulation from the same code that runs on real hardware greatly simplifies the development process. TOSSIM supports several realistic radio-propagation models and has been validated against real deployments for several applications. The design of TOSSIM allows the same code that is run on real hardware to be tested in simulation at scale. TOSSIM captures the behavior and interactions of networks of thousands of TinyOS motes at network bit granularity, [11].

TOSSIM provides run-time configurable debugging output, allowing a user to examine the execution of an application from different perspectives without needing to recompile. TOSSIM also incorporates TinyViz, a Java-based graphic user interface (GUI) that allows for visualization and control of the simulation as it runs, inspecting debug messages, radio and UART packets, and so forth. It has a set of plugins like debug messages, radio model, ADC readings, etc which provide the desired functionality.

Although TOSSIM captures TinyOS behavior at very low level, it does not model power consumption for motes. This is because it does not model CPU execution time, and thus, cannot provide accurate information for calculating CPU energy consumption, [11]. The authors of [8] designed a tool called PowerTOSSIM to measure the CPU cycles and power consumption for a particular node running a specific application. This tool is integrated in all the versions of TinyOS after version 1.1.9.

5 Implementation and results

We used TinyOS 1.1.14 the most recent version of TinyOS as of this writing. The component TinySecM.nc, present in the library, is the main component for TinySec implementation. It contains the code for cryptographic operations. This code was modified to include the key update code. TinySecMode interface contains commands to specify the TinySec transmission and reception modes. New commands were added to set the key update mode. Three different applications were used to obtain results for memory overhead, CPU cycles, and network behavior for the proposed scheme.

The application called TestTinySec was used as a reference for memory overhead evaluation. This application is pre-built in TinyOS and comes with its download package. The sample application is compiled for the mica2 mote. The base application without the key update code serves as reference and the results for ROM and RAM usage are tabulated Table 1 below. The second column shows the memory values after the addition of the key update code. The key update code adds 1.66 % overhead for ROM and 0.347 % overhead for RAM per node.

Table 1: Summary of Memory Overhead

Memory	Without Key Update code	With Key Update code	Difference	% increase
ROM	24164 bytes	24566 bytes	402 bytes	1.66 %
RAM	865 bytes	868 bytes	3 bytes	0.347 %

To compute the CPU cycle and measure power, we used the powerTOSSIM tool. We determined that PowerTOSSIM does not provide for measuring of the CPU cycle values and power consumption between the two modes of TinySec (authentication and authentication with encryption). The AE mode involves encryption which is expected to consume much higher energy, however, the results showed no difference between the two modes. Also, the powerTOSSIM paper, [8] does not include TestTinySec in the list of applications experimented with.

To measure the CPU cycle count for our scheme, we modified the Blink application which just includes the code for toggling the red LED. The code required for key update was put in a task and added to the application. During the simulation, the task was called at each timer firing event, set to fire every second. To get the difference, the application was compiled and run for two cases:

- Measuring CPU cycle count without invoking the task
- Measuring CPU cycle count with invoking the task at each timer firing event.

The simulation time was 60 seconds. Table 2 shows the results. Therefore, CPU cycles for one key update operation = $(10310 - 991.5) / 60 = 155.308$ cycles. The number of CPU cycles required is comparatively less than that of other

cryptographic operations, for example, the encryption which takes 103756 cycles, [8]. This shows the efficiency of the scheme.

To simulate the network behavior, we designed a new application having multiple nodes and a base station. TOSSIM programs all the simulated nodes with the same code (both for base station and normal node) but they could be distinguished at runtime by their node number or node ID. Here *node 0* is the base station and all the other nodes sense the environment and send their readings to the base station.

<pre> task RotateKey() { Instructions for key update; } </pre> <p>Figure 3: Representation of task added</p>	<p>Table 2: CPU cycle measurement</p> <table border="1"> <thead> <tr> <th>Without task</th> <th>With task</th> </tr> </thead> <tbody> <tr> <td>991.5</td> <td>10310.0</td> </tr> </tbody> </table>	Without task	With task	991.5	10310.0
Without task	With task				
991.5	10310.0				

Each node has two sensors attached to it; temperature and light (photo). They sample the readings of these sensors alternately and send them to the base station at each timer firing event. The data is sent after encryption. The base station sends a key update request periodically (after every 20 timer firing events) and all the nodes are expected to update their keys and send the future data encrypted with the new key. Figure 4 shows the screenshot of the simulation using TinyViz. Here the blue circle indicates broadcasting a message and the magenta arrows show one-to-one communication. The base station broadcasts the key update message periodically, and all the other nodes send their readings to the base station. The simulation of figure 4 is for single-hop network where all the nodes are in a single cell. Figure 5 shows the implementation for multi-hop environment. Here, the network consisting of 3 nodes. Node 0 acts as base station and broadcast the key update request. Node 1 is in the range of node 0 but node 2 is not. Due to this, only node 1 hears the key update request. It first updates its key and then propagates the request to node 2. Node 2 then updates its key.

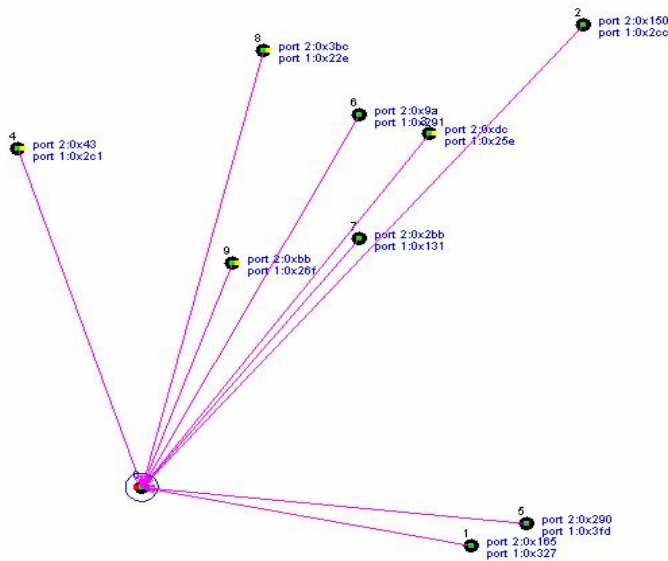


Figure 4: TinyViz Screen-shot showing the behavior of a 10-node network for the key update process



Figure 5: Simulation of multi-hop environment showing the key update process

6 Related Work

Some researchers have addressed the key update issue prior to this research. Malan et. al.[12] proposed to use Elliptical curve cryptography (ECC) for key update. Being a public key cryptography scheme, ECC is more secure. However, it consumes a considerable amount of computational resources and memory. It requires 34.1 KB of RAM which is 26.5% of the total RAM space available. Also, the energy required is 0.9 joules and the computation time is around one minute. Watro et. al. [13] show the successful implementation of key update for two nodes using Diffie-Hellman key exchange. Their results show that Diffe-Hellman is also an expensive method for key update in sensor networks. It requires around 2 minutes of computation time, 12 KB of ROM and 1 KB of RAM. In their complete robust protocol LEAP, [5], Jajodia et. al. use four different keys to support the security features of authentication, encryption and key update. LEAP too is an expensive protocol.

7 Conclusion

In this paper we present an efficient key update scheme for updating TinySec keys without any significant computation effort and storage requirements for sensor networks. The scheme is equally secure compared to other key update mechanisms such as ECC, TinyPK, LEAP etc. The aforementioned schemes require more computational time and power and consume much more of the scarce sensor node resources. Our scheme bolsters what TinySec offers in the event that the network lives longer by allowing key updating at very low power and 1.66% memory overhead with no bandwidth penalties. Simulations performed on the most successful link layer protocol show promise for this approach.

8 References

- [1] Chris Karlof and David Wagner, "Secure Routing in Wireless Sensor Networks: Attacks and Countermeasures", *IEEE International Workshop on Sensor Network Protocols and Applications*, 2003
- [2] Elaine Shi and Adrian Perrig, "Designing Secure Sensor Networks", *IEEE Wireless Communications*, pp 38-43 December 2004
- [3] Adrian Perrig, Robert Szewczyk, Victor Wen, David Culler, J. D. Tygar, "SPINS: Security Protocols for Sensor Networks", *Mobile Computing and Networking*, 2001 Rome, Italy
- [4] Chris Karlof, Naveen Sastry and David Wagner, "TinySec: A Link Layer Security Architecture for Wireless Sensor Networks", *SenSys'04*, November 3-5, 2004, Baltimore, Maryland, USA
- [5] Sencun Zhu, Sanjeev Setia, Sushil Jajodia, "LEAP: Efficient Security Mechanism for Large-Scale Distributed Sensor Networks", *Tech-Report (ACM)*, Aug.2004
- [6] Seyit A Camtepe, Bulent Yener, "Key Distribution Mechanisms for Wireless Sensor Networks", *Technical Report TR-05-07, March 23, 2005*, Department of Computer Science, Rensselaer Polytechnic Institute.
- [7] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K.Pister, "System architecture directions for networked sensors", *In Proc. of ASPLOS IX*, 2000.
- [8] Victor Shnayder, Mark Hempstead, Borrong Chen, Geoff Werner Allen, and Matt Welsh, "Simulating the Power Consumption of LargeScale Sensor Network Applications", *SenSys'04*, November 3-5, 2004, Baltimore, Maryland, USA
- [9] Ernest F. Brickell, Dorothy E. Denning, Stephen T. Kent, David P. Maher and Walter Tuchman, "*SKIPJACK Review - Interim Report*", July 28, 1993
- [10] Philip Levis, Nelson Lee, Matt Welsh, and David Culler, "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications", *In Proceedings of the first ACM Conference on Embedded Networked Sensor Systems (SenSys) 2003*, Nov. 2003
- [11] Philip Levis and Nelson Lee, "TOSSIM: A Simulator for TinyOS Networks", <http://www.tinyos.net/tinyos-1.x/doc/nido.pdf>
- [12] David J. Malan, Matt Welsh, Michael D. Smith, "A Public-Key Infrastructure for Key Distribution in TinyOS Based on Elliptic Curve Cryptography", *IEEE International Conference on Sensor and Ad Hoc Communications and Networks (SECON04)*, 2004
- [13] Ronald Watro, Derrick Kong, Sue-fen Cuti, Charles Gardiner, Charles Lynn1 and Peter Kruus, "TinyPK: Securing Sensor Networks with Public Key Technology", *SASN'04 (ACM)*, October 25, 2004, Washington, DC, USA.