A TPM-enabled Remote Attestation Protocol in Wireless Sensor Networks

Hailun Tan[1] Wen Hu[2] Sanjay Jha[1]

[1] University of New South Wales, Australia
{thailun,sanjay}@cse.unsw.edu.au
[2] ICT Centre, CSIRO
wen.hu@csiro.au

Technical Report
UNSW-CSE-TR-1105
May 2011

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

Given the limited resources and computational power of current embedded sensor devices, memory protection is difficult to achieve and generally unavailable. Hence, the software run-time *buffer overflow* that is used by the worm attacks in the Internet could be easily exploited to inject malicious codes into Wireless Sensor Networks (WSNs). Previous software-based remote code verification approaches such as SWATT and SCUBA have been shown difficult to deploy in recent work. In this paper, we propose and implement a remote attestation protocol for detecting unauthorized tampering in the application codes running on sensor nodes with the assistance of Trusted Platform Modules (TPMs), a tiny, cost-effective cryptographic micro-controller. In our design, *each* sensor node is equipped with a TPM and the firmware running on the node could be verified by the other sensor nodes in a WSN, including the sink. Specifically, we present a hardware-based remote attestation protocol, discuss the potential attacks an adversary could launch against the protocol, and provide comprehensive system performance results of the protocol in a multi-hop sensor network testbed.

1 Introduction

Current research on security for Wireless Sensor Networks (WSNs) focuses on attacks such as communication channel jamming [25], countering attacks on routing protocols [10], providing attack-resistant code dissemination [3, 23] and securing node localization [27].

A potentially more severe attack called *sensor worm attack* has yet to be fully studied. In this attack, an attacker would first attempt to discover an exploitable *software vulnerability*, for example by examining a physically captured sensor node. These software vulnerabilities are commonly found in popular sensor network operating system libraries such as TinyOS [6, 1]. By exploiting the vulnerabilities, the attacker can then inject packets, carrying the malicious codes, into the network, furthermore, by exploiting the popular features in bootloader such as over-the-air-programming [9]. Hence, a malicious program could hijack execution by injecting itself into the program memory and self-propagate to other nodes by such a buffer overflow event [6, 19, 18], which is further explained in Section 2.1.

Therefore, it is very important to have a protocol to verify the trustworthiness of a remote sensor node for many mission-critical WSN applications such as e-health and critical infrastructure monitoring. The *remote attestation* aims at verifying the program flash memory of the sensor nodes. Recovery measures are taken if an unauthorized alteration in the program code has been discovered (e.g., exclusion or reprogramming of the corresponding tampered sensor nodes). Figure 1.1 illustrates a generic architecture of a remote attestation protocol. The challenger generates a challenge and sends it to the sensor node to be attested (attestation responder in Figure 1.1). Upon receiving this challenge, the sensor node will check the corresponding firmware, construct the attestation response and returns the response, which is associated with the challenge content. The sink can verify this response based on the challenge it generated earlier and the expected firmware content. The attacker might keep a copy of the correct firmware simply to answer the challenge from the sink (i.e., step 3 in Figure 1.1) and make the malware practically run on the sensor nodes. Therefore, most software-based attestation protocols such as SWATT [19] and SCUBA [18] sets a timer right after it sends out the challenge. If there is a timeout before a response is received, the sink will suspect that the sensor node to be attested has been compromised. A recent research has demonstrated that these secure time-based attestation schemes (namely SWATT and SCUBA) are very difficult to design and implement correctly in practice [1].

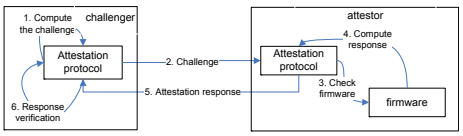


Figure 1.1: The architecture overview of remote attestation protocol: the numbers represent the temporal ordering of events.

Generally the attestation techniques based on additional hardware such as Trusted Platform Modules (TPMs) has been considered infeasible in WSNs because of its size, financial cost and energy overhead. However, a recent work

has shown that a TPM chip (Atmel $AT97SC3203S$) costs $4.5 when ordered in large-quantity, which is less than 5% of the cost of a typical sensor node ($100). Furthermore, the actual TPM chip is small (6.1x9.7mm),which is less than 2% of the area of a typical sensor node [**?**].

In this paper, we present the design and implementation of a new hardware-based remote attestation protocol to detect unauthorized alteration in the application code, particularly due to the sensor worm attacks. Our evaluation results based on a multi-hop testbed show that the energy consumption, attestation latency and code size of proposed TPM-enabled remote attestation protocol are viable in WSNs .

The contributions of this paper include:

- **A new approach for remote attestation in WSNs**
  Most of the previous work addressing remote atestation in WSNs is software-based [16, 19, 18, 20]. It typically assumes that sensor nodes do not incorporate extra security hardware. Therefore, these software-based approaches depend on the strict response time measurement to ensure the correct attestation responses are not forged by the attackers. Recent research has shown these software-based attestation protocols could be beaten by generating the correct attestation response quicker than expected [1]. Though some of the previous approaches adopt a TPM to detect node compromise, the TPM is used for the cluster head only rather than each sensor node [11]. Therefore, only the cluster heads can be attested in [11]. Moreover, there are analytical models for power consumption and storage overhead in [11] rather than empirical evaluation. In our scheme, since each node is equipped with TPM, each node can challenge its neighbors and be verified by other nodes. To the best of our knowledge, the feasibility of equipping each sensor node with the TPM for memory protection and associated protocols and mechanisms to support this is first studied in this paper. In addition, we further evaluated our protocol on a testbed and studied how our protocol can effectively counter the recent attacks, which have beaten the existing software-based attestation protocols.

- **Monitor of platform configuration in WSNs**

  As there are several Platform Configuration Registers (PCRs) in a TPM (see Section 3.1 for the detailed description of PCRs), a hash of the environment variables could be stored in a PCR by the bootloader each time a sensor node is rebooted. If the firmware is updated by the over-the-air reprogramming from bootloader, any modification in the sensor platform configuration would be embodied in the different value of the PCR when the bootloader is invoked. Though PCRs have been used to monitor the platform configuration on PCs with TPM equipped, where the resource (e.g., power consumption, memory overhead, etc.) is not an issue. The platform configuration monitor in WSN has not been explored before. To our knowledge, this paper is the first to employ the platform configuration monitor using lightweight mechanisms suitable for remote attestation in WSN.

- **The new countermeasures against TPM-related attacks**

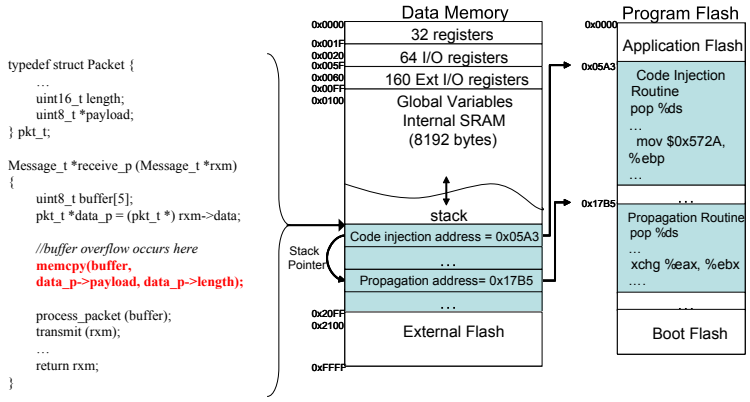
Given the limited resources in WSNs, some TPM-related attacks (e.g., Cuckoo attack [15] and TPM reset Attack [12]) that can be defended against in PC will pose a threat in sensor nodes equipped with TPM. In our hardware-based attestation protocol, these attacks are studied and tackled with the new countermeasures, discussed in Section 4.3 and 4.4.

The rest of this paper is organized as follows. The attacker model, background and our assumptions are presented in Section 2. The design and implementation of the proposed hardware-based remote attestation protocol is described in Section 3, followed by the security analysis in Section 4 and the respective performance evaluation in Section 5. The related work is surveyed in Section 6 and the paper is concluded in Section 7.

2 Attacker Model, Background and Assumptions

2.1 Attacker Model

In this paper, the attacker is able to inject the malicious codes into the program flash through *buffer overflow*. The malicious codes can self-propagate to other nodes by exploiting the over-the-air reprogramming features from bootloader, which will be introduced in Section 2.2. This attack is known as *sensor worm attack*. In this attack, an attacker would first need to discover an exploitable *software vulnerability*, for example by examining a physically captured sensor node. One type of software vulnerability is a spot where a *buffer overflow* could occur within the source code. Figure 2.1 illustrates this attack on the Atmel ATmega1281 micro-controllers, where the data memory and program flash is separated from each other. The code in this example is to process a packet upon reception ($receive\_p$ function in Figure 2.1(a)), and then broadcast this packet to downstream nodes ($transmit(rxm)$ in Figure 2.1(a)). However, the array boundary can be overflown (i.e., $data\_p \rightarrow length > 5$ in *memcpy* function in Figure 2.1(a)) and a part of memory gets overwritten inappropriately (see Figure 2.1(b)). With the memory overwritten with the undesired code, the attacker attempts to inject packets (e.g., code injection routine in program address 0x05$A3$ is invoked in Figure 2.1(a)), carrying the malicious codes into the network (e.g., propagation routine in program address 0x17$B5$ is invoked in Figure 2.1(a)). This sensor worm attack has been further studied and implemented in [6]. Though the malicious code injection can be detected by remote attestation protocols, the attacker can still circumvent these detection techniques by constructing the correct attestation responses from the legitimate code image before it is overwritten by the malicious one. Therefore, most software-based attestation protocols [19, 18, 21] depends on the strict attestation time measurement to determine whether the correct attestation response is replayed or forged. This strict time measurement is difficult to implement given the network delay in WSNs. Moreover, recent research has demonstrated that a new attack called *Rootkit-based attack* can practically beat the strict response time measurement in these software-based attestation protocols [1]. In our protocol design, the Rootkit-based attack is considered and can be detected. The relevant discussion is presented in Section 4.2.



(a) C code to be exploited in sensor application and the corresponding memory allocations.

```
uint8_t payload[] = {

    0x00, 0x01, 0x02,0x03, 0x04,    //fill the buffer
    0x05, 0xA3,                     //program address to copy malware
    0x57, 0x2A,                     //address to place malware

    MALWARE DATA,

    0x00,                           //padding
    0x17, 0xB5,                     //program address to propagate malware
    …
    0x00, 0x00                      // reboot the sensor node
}
```

(b) Packet payload to inject the malicious code.

Figure 2.1: A buffer overflow example on memory structure of Atmel ATmega1281 micro-controllers (Harvard architecture).

2.2 Background

Architecture of microcontroller in WSNs

Depending on the different architectures of microcontrollers, the efforts to inject the malicious codes can vary. Since data memory and program flash are allocated together in Von Neumann architecture such as MSP430 microcontroller used in the Tmote, the attacker can overwrite the program address space through exploiting the buffer overflow in data memory directly. In order to cover the security risk in the design of Von Neumann architecture, the Harvard architecture is adopted in *all* AVR micro-controllers (e.g., Atmel ATmega128 in micaZ). The Harvard architecture has three physically separate internal memories: EEPROM memory, program flash and data memory (SRAM).

Though the program flash, data memory and EEPROM are separated from one another, the microcontroller lacks a Memory Management Unit (MMU). No memory protection is available for data memory address access. As a result, the entire data memory space, including the registers, I/O interface and stack are addressable.

Another feature of the Harvard architecture is the memory access control. The data memory is writable through the application while the program flash is read-only unless the specific hardware programmer is used. Hence, it is widely accepted that an attacker could not easily inject the malicious code image into

program flash through, for example, buffer overflow.

Bootloader and Over-the-air-programming

Although the program flash is read-only in Harvard architecture unless the hardware programmer is used, there are many scenarios where the running applications on sensor nodes need to be updated or changed after deployment. For example, if a software error is detected, the application code in the program flash should be updated via wireless communication since a WSN may have thousands of nodes, it is not practical (and often impossible) for the network administrator to reprogram each sensor node *in situ* with a hardware programmer. Hence, a code update mechanism over-the-air is a critical requirement for the reliability and survivability of a large-scale WSN. This requirement is addressed by *bootloader*, which is a piece of monolithic code residing in a reserved area of program flash. The bootloader is installed into the program flash before node deployment, through a conventional hardware programmer. However, what the bootloader does is different from any conventional application. Once installed, the bootloader will start listening to the incoming messages through the radio interface to obtain the application update, and store them in the external flash given the limited size of program memory. After the complete application update is received, it is copied from external flash to data memory, from which the bootloader could transfer the application update to program flash through Store to Program Memory (SPM) command in the AVR assembly language. The sensor node starts to execute the new application code after reboot, triggered by the bootloader. One important prerequisite is that the bootloader could not be overwritten over-the-air as it is required to receive the application update over-the-air. Therefore, it is possible for the attacker to exploit the self-programming routine in the bootloader so that the malicious code can be copied from the data memory to program flash [6].

2.3 Assumptions

In this paper, we assume that an attacker could not physically compromise a large number of sensor nodes in large-scale deployment given that it would require the attacker's hardware (e.g., a PC or laptop with JTAG) to physically enter the deployment region. This increases an attacker's risk of being detected. However, we assume that the attacker could physically capture a small number of sensor nodes through use of a PC-class device and some dedicated, specialized hardware such as JTAG [8] equipment. Consequently, software vulnerabilities in the application code may be discovered by an attacker. Malware is assumed to be self-propagating (i.e., after installation, a contaminated node can invoke the code injection routine to propagate the code image to its downstream peers). In this way, an attacker could compromise the whole network by physically capturing few sensor nodes only.

The code injection routine is one of the functionalities of the *bootloader*. Henceforth, we assume each sensor node is configured with a *bootloader*. This assumption is realistic given the requirement for the over-the-air code update mechanism described in Section 2.2. Given the characteristics of the bootloader in Fleck node used for prototype of our implementation, we do not require the remote code update protocol, such as Deluge [9], to be available in sensor nodes

in the rest of this paper. However, if such a protocol exists, we assume that it is secure (i.e., the code update image is authenticated [3, 23]) since the buffer overflow occurs after packet authentication and before the new program image is executed.

3 TPM-enabled Remote Attestation

3.1 Trusted Platform Module Basics

The objective of a TPM is to provide a hardware-based *root of trust* for a computing system. We provide a brief introduction of some TPM features which is important to understand our work in Trusted Computing standard specification [14]. The following three key factors are utilized in our attestation protocol design:

- *Cryptographic operating engine*

  In TPM, a range of the cryptographic operations are available, including RSA engine for signature generation and message decryption, Secure Hash Algorithm (SHA) Engine, Random Number Generation (RNG). The security design in WSN with TPM could be improved over the software-based security design in the following two aspects. Firstly, every TPM is programmed with a unique RSA key pair and the private part of the RSA key pair never leaves the non-volatile storage area (i.e., protected memory) of TPM. Despite sensor nodes being captured by an adversary, the private part of the RSA key would not be available to the adversary for further attacks. Secondly, the cryptographic executions in TPM are much more efficient than the software-based solutions [24, 13] in terms of the operation time and power consumption[**?**].

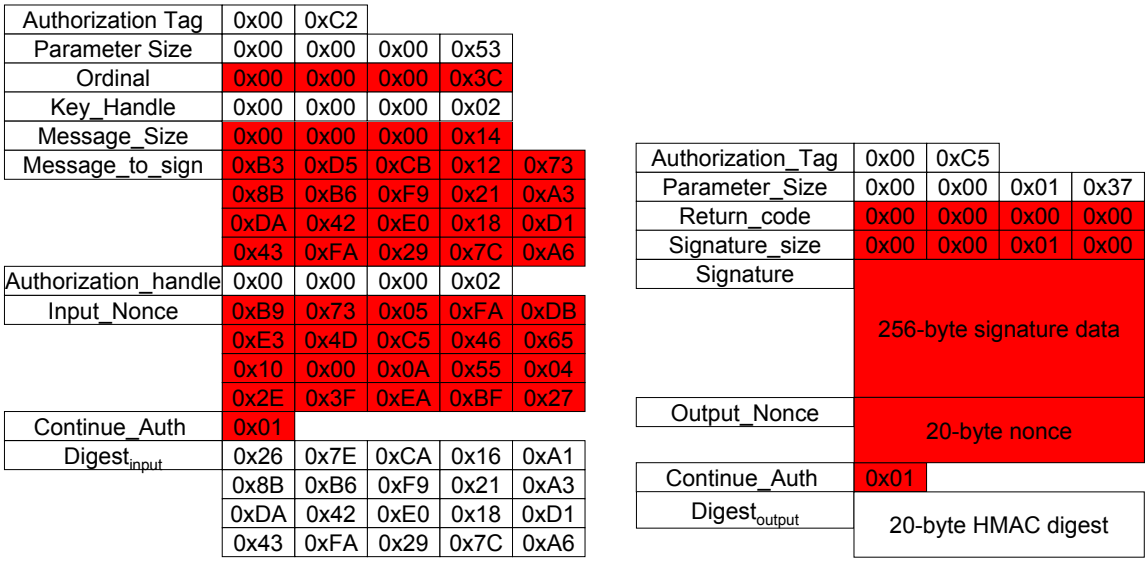
- *Platform Configuration Register(PCR)*

  TPM contains a number of (usually 16) platform Configuration Registers (PCRs). The content stored in each PCR is a digest of messages related to the platform environment (or some other integrity-sensitive messages). PCRs are located in the non-volatile storage area (shielded internal memory slots) and hence could not be directly tampered with.

- *Sealed storage*

  Sealed storage is a special type of message encryption/decryption provided by TPM. The sealed message is not only associated with the encryption key, but also bound to the selected PCR values of platform sealing the message. The seal operation indicates that the sealed message could be retrieved not only with the right key for decryption but also in the exactly same platform environment (i.e., a *trusted* environment) where this message is sealed. With the sealed storage, even though adversary might intercept the sealed message and retrieve the key for encryption during a node compromise, it is difficult for adversary to completely mimic the *trusted* environment to *unseal* the message.

  Since almost all existing security schemes in WSNs will require some information to be preloaded into the sensor nodes before deployment, how



(a) The TPM_Sign command.

(b) A successful TPM_sign response.

Figure 3.1: TPM_Sign command and its response: the colored data fields are included in the computation of authorization digest.

to secure these preloaded secrets from being known to attackers post deployment introduces a new challenge in the security scheme. In security, anything that one person designs and makes can always be cracked by another person given sufficient time, costs and opportunities. The goal of security is to design a *root of trust* whose price of cracking is much higher than the information it reveals. In the context of sensor network, these preloaded secrets are the root of trust, which should be protected with the highest security level so that the attackers could not learn about these preloaded secrets on other sensor nodes through a few physically compromised sensor nodes. The preloaded secret in the secure scheme can be *sealed* with the root storage key, which is stored in TPM internally. Since TPM is a piece of cryptographic hardware, these preloaded secrets will be shielded from the attackers and used as the *root of trust*.

The TPM board used in this paper is Atmel $AT97SC3203S$ TPM chip while the sensor hardware adopted is a custom built WSN node called Fote.

In our design, several TPM commands will be used to construct the attestation responses and verify these responses. According to the security level, they are categorized as either *unauthorized commands* or *authorized commands*:

- *Unauthorized commands* are those which could be executed without authorization as long as the command format complies the Trusted Computing standard [14]. As long as TPM is in the right state, these commands can be executed successfully (e.g., $TPM\_Startup$ can successfully boot up the TPM if TPM is in shutdown state). Since they are not bound to any authorized entity (e.g., the asymmetric RSA key pair), They do

not require a nonce in the incoming TPM messages to generate output. The unauthorized TPM commands used in our scheme are $TPM\_Startup$, $TPM\_GetRandom$, TPM_Hash, $TPM\_Turnoff$,$TPM\_PcrExtend$, $TPM\_VerifySignature$.

- *Authorized commands* are those which require certain authorization before they could be executed. An input nonce is required. There will be an output nonce, a 20-byte Hashed M essage Authentication Code (HMAC) result as authorization digest, together with the output result. The authorization digest is derived from the input nonce, output nonce and the output result. For example, Figure 3.1 illustrates the formats of the input/output messages for an authorized TPM command - TPM_Sign.

  For the input command, a hashed result called *payload digest* (denoted as P_{input}) is firstly computed as follows

$$P_{input} = H(Ordinal||Message\_Size \\ ||Message\_to\_Sign) \tag{3.1}$$

  where $H(.)$ represents the $SHA-1$ hash function [4], $A||B$ denotes message A is concatenated with B and Ordinal is a unique ID for each TPM command [14].

  The authorization digest of TPM command ($Digest_{input}$ in Figure 3.1(a)) is computed as

$$Digest_{input} = HMAC(entity\_secret, P_{input}|| \\ Input\_Nonce||Continue\_Auth) \tag{3.2}$$

  where P_{input} is from (3.1), $HMAC()$ is the HMAC function [4] and *entity_secret* is the HMAC key input. The *entity_secret* is a 20-byte value bound to the TPM entity whose authorization is required in the corresponding TPM command. In the example illustrated in Figure 3.1, the *entity_secret* is referred to as the usage secret of the RSA key pair, which is sealed in the TPM internally.

  For the TPM response, the payload digest is computed as

$$P_{output} = H(Return\_Code||Signature\_Size \\ ||Signature) \tag{3.3}$$

  where *Return_Code* is a 32-bit value indicating the execution state of the TPM command. If the command is executed successfully without any error, *Return_Code* will be zero, as is shown in Figure 3.1(b). Otherwise, *Return_Code* will be some non-zero value, which denotes a specific execution error [14].

  The authorization digest of TPM response ($Digest_{output}$ in Figure 3.1(b)) is computed as

$$Digest_{output} = HMAC(entity\_secret, P_{output}|| \\ Input\_Nonce||Output\_Nonce \\ ||Continue\_Auth) \tag{3.4}$$

where the entity secret is the same as the one in equation (3.2) and $Continue\_Auth$ is a one-byte flag indicating whether the current authorization session continues or not.

Therefore, the nonce pair (i.e., Input_Nonce and Output_Nonce in (3.4)) is used to ensure that the output result is generated from a genuine TPM and it could not be replayed by the verification with the output nonce and authorization digest. The authorized commands used in our scheme are $TPM\_Loadkey$, TPM_Seal, $TPM\_NVWriteValue$, TPM_Unseal, TPM_Sign, $TPM\_NVReadValue$. In most cases, the output result of the TPM authorized commands is requested locally (i.e., requested by the corresponding sensor node), assuming that the TPM board never leaves the node. Thus, unless specified, the authorization digest and the input/output nonce are not shown in the Algorithm 1 - 2.

3.2 Design and Implementation

In the following description, we assume that node A issues an attestation challenge (i.e., node A is challenger) and node B is attested (i.e., node B is attestation responder). The attestation protocol consists of three stages: initialization, bootloader stage and application stage. Both node A and node B follow the same procedure at initialization and bootloader stage. Only the code for one node is shown in Algorithm 1-2 for demonstration purposes. These three stages are discussed in Section 3.2, 3.2 and 3.2, respectively. The multi-hop expansion of this protocol is further discussed in Section 4.5.

Initialization

The initialization phase precedes deployment and hence it is safe to assume that nodes have not been compromised. The system administrator loads the bootloader into sensor node and pre-configures the TPM board (e.g., load the RSA key pairs into TPM).

Before two nodes could communicate with each other securely, a shared secret (K_{AB}) is required (line 1 in Algorithm 1). There exists the classic approach to establish the shared secret before two nodes communicate with each other [2][1]. In addition, a secret between node B and the base station (K_{BS}) is loaded into bootloader code of node B (line 2 in Algorithm 1). This secret (K_{BS}), available at bootloader stage, is used only to generate the correct attestation response that could not be forged at application stage. K_{AB} is available at the application stage to ensure that the response is from the non-volatile area.

After these shared secrets are established, TPM is started (line 3 in Algorithm 1) and the RSA key pair to seal K_{AB} is loaded into TPM (line 4 in Algorithm 1). In our scheme, the binary code of the bootloader is hashed and extended into the PCR (line 5, 6 in Algorithm 1). This PCR value would be used to seal the shared secret (K_{AB}, line 7 in Algorithm 1). The reason why the hashed result of the bootloader code rather than that of the application code is adopted is that the bootloader cannot be changed after deployment by

[1]The shared secret establishment is beyond the scope of this paper. Interested readers could refer to [2] for further details.

over-the-air reprogramming while the application might be either updated by an authorized user or altered by the attacker. The hashed result of the bootloader content could serve as the first defense line for the attestation protocol. If the bootloader is changed at a later stage, the unseal command would fail as the associated PCR_2 is different, resulting in the attestation failure state (see Figure 3.3).

Algorithm 1 Initialization

Require: The following operations are carried out before deployment.
Ensure: The hashed values of bootloader content (M_b) is put into Platform Configuration Register

1: A and B establish the shared secret K_{AB}
2: B and base station establish the shared secret K_{BS}
3: $A \rightarrow TPM_A : TPM\_Startup$
4: $A \leftarrow TPM_A : TPM\_loadkey$ for $SEAL\_KEY_A$
5: $A \leftarrow TPM_A : h_b \leftarrow TPM\_HASH(M_b)$
6: $A \leftarrow TPM_A : V_{PCR_2} \leftarrow TPM\_PcrExtend(h_b, PCR_2)$
7: $A \leftarrow TPM_A : E_{SEAL\_KEY_A}(K_{AB}) \leftarrow TPM\_SEAL(SEAL\_KEY_A, K_{AB}, PCR_2)$
8: $A \rightarrow TPM_A : TPM\_turnoff$

Bootloader stage

At this stage, the bootloader is running but the application (M_p in Algorithm 2) is not yet loaded into the program flash. The node to be attested will be rebooted with the installed application after the initial execution of the bootloader, which will subsequently listen to the radio interface for any code update. We could assume the node is vulnerable to physical capture but not susceptible to malware injection as the bootloader could not be overwritten over-the-air.

Algorithm 2 shows operations performed in our attestation protocol at bootloader stage (before listening to the radio interface for code updates over-the-air). A hash of the application code, concatenated with the shared secret between the node and base station, is generated (line 2 in Algorithm 2) after TPM is started up (line 1 in Algorithm 2). This hashed value is extended into Platform Configuration Register (PCR) (line 6 in Algorithm 2). The extended result will then be written into the non-volatile storage area (i.e., NV_Area1 in Algorithm 2) within TPM (line 7 in Algorithm 2). The hashed result of the bootloader code, extended into PCR_2, would be written into the different non-volatile storage area (line 5 in Algorithm 2) to unseal the shared secret at the application stage.

Application stage

At this stage, the application is running on the node B once the nodes are rebooted after bootloader stage. We assume that the attacker can physically capture a small number of sensor nodes, exploit a software vulnerability and launch the code injection attack over the air.

The sequence of the critical events during the application stage are shown in Figure 3.2. Node A will first generate a 20-byte random nonce, N_A with $TPM\_GetRandom$. Then it will try to unseal the shared secret between node A and node B (K_{AB}). If the unseal operation is unsuccessful, the attestation

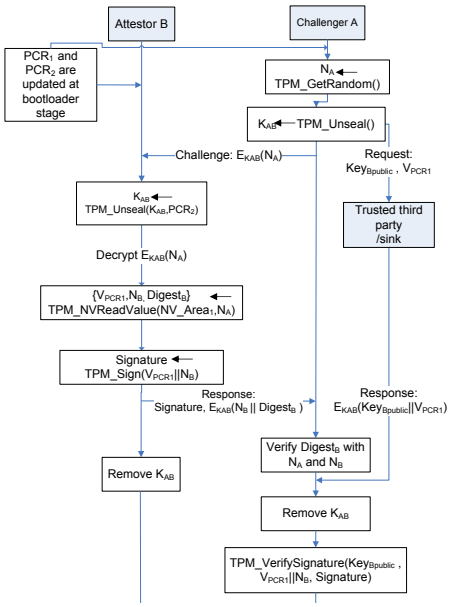


Figure 3.2: The remote attestation process at application stage: node A is a challenger to verify the attestation response from node B.

process fails and stops (see Figure 3.3(a)). Otherwise, N_A will be further encrypted with K_{AB} (i.e., $E_{K_{AB}}(N_A)$ in Figure 3.2, symmetric encryption), which is the challenge. Then node A will send out the challenge and transit to "wait for response" state (see Figure 3.3(a)). During this period, node A requests the expected response and the public key of node B (i.e., V_{PCR_1} and $Key_{B_{public}}$ in Figure 3.3(a) or Figure 3.2) from the third-party trusted entity (e.g., base station or sink). Upon receiving this request, the sink or third-party will send a response with requested content, encrypted by the shared secret between the attestation responder (node B) and the challenger(node A) (i.e., $E_{K_{AB}}(Key_{B_{public}}||V_{PCR_1})$ in Figure 3.2).

Upon receiving the challenge from node A, node B would unseal K_{AB}. Similarly, if the unseal operation fails, it indicates that the PCR_2 is different from the version when K_{AB} is sealed. The attestation process stops and node B sends back the "attestation fails" message to node A (see Figure 3.3(b)). Otherwise, node B decrypts the challenge to retrieve N_A. N_A becomes the input nonce to the $TPM\_NVReadValue$, which yields N_B (output nonce),
$Digest_B$ (authorization digest) and V_{PCR_1}, containing the information on the content of program memory (see line 2, 6 in Algorithm 2). After that, node B will construct the challenge response by signing the concatenation of V_{PCR_1} and N_A (i.e., $Signature \leftarrow TPM\_Sign(V_{PCR_1}||N_A)$ in Figure 3.2) and encrypting the concatenation of N_B and $Digest_B$ (i.e., $E_{K_{AB}}(N_B||Digest_B)$ in Figure 3.2, symmetric encryption). The signature and the encryption results are sent back to node A for verification. Then node B would remove K_{AB}(Figure 3.2) and return to the idle state (see Figure 3.3(b)).

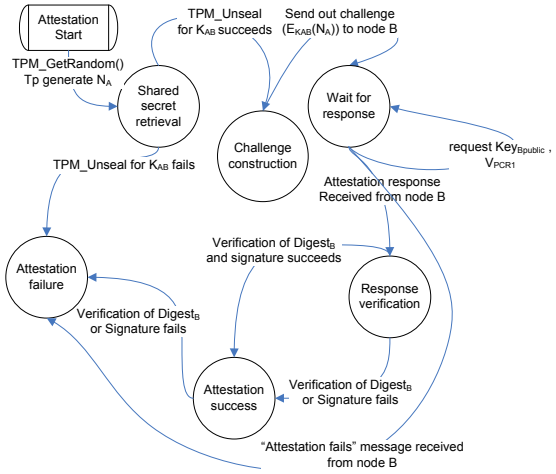
On receiving the response from node B, node A would first decrypt $E_{K_{AB}}(N_B||Digest_B)$ and verify the authorization digest ($Digest_B$) with N_A and N_B. Only if the

Algorithm 2 Remote attestation protocol on node B at bootloader stage

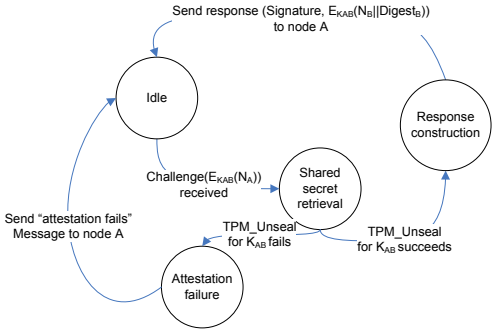
Require: The bootloader itself could not be overwritten by over-the-air programming. The shared secret between base station and node B (K_{BS}) is available.

Ensure: the hashed values of application content (M_p), the ending program address of the application content ($Address_p$) and bootloader content (M_b) are put into the Platform Configuration Registers.

1: $B \rightarrow TPM_B : TPM\_Startup$
2: $B \leftarrow TPM_B : h_p \leftarrow TPM\_HASH(M_p||K_{BS})$
3: $B \leftarrow TPM_B : h_b \leftarrow TPM\_HASH(M_b)$
4: $B \leftarrow TPM_B : V_{PCR_2} \leftarrow TPM\_PCREXTEND(h_b, PCR_2)$
5: $B \rightarrow TPM_B : TPM\_NVWriteValue(V_{PCR_2}, NV\_Area2)$
6: $B \leftarrow TPM_B : V_{PCR_1} \leftarrow TPM\_PCREXTEND(h_p, PCR_1)$
7: $B \rightarrow TPM_B : TPM\_NVWriteValue(V_{PCR_1}, NV\_Area1)$
8: $B \rightarrow TPM_B : TPM\_turnoff$



(a) The state diagram of node A (challenger).



(b) The state diagram of node B (attestation responder).

Figure 3.3: The state diagrams of node A (challenger) and node B (attestation responder).

verification on $Digest_B$ succeeds will node A verify $Signature$ with V_{PCR_1} and $Key_{B_{public}}$, retrieved from the trusted third-party or sink. If the signature verification also passes, the attestation process succeeds. Otherwise, the attestation process fails and node B might have been compromised.

4 Security Analysis

The most common attack against the remote attestation is referred to as a Time-Of-Check-To-Time-Of-Use (TOCTTOU) attack. In this attack, the attacker memorizes the correct code image after node compromise. Upon being attested, the attacker could just send back the response from the correct code image to the challenger (time-of-check case). However, the actual program flash to be executed is different when it is invoked (time-of-use case), which is presented in Section 4.1. The rootkit-based attack proposed by Castelluccia et al. [1] against

challenge-response attestation protocol is investigated in Section 4.2. The TPM-related attacks (Cuckoo attack [15] and TPM reset attack [12]) are investigated in Section 4.3 and 4.4, respectively. The man-in-middle attack against the multi-hop attestation discussed in Section 4.5. Some additional implementation issues are described in Section 4.6.

4.1 TOCTTOU Attack

Three cases must be considered in the TOCTTOU attack:

- The correct code image is copied to another location in program flash memory and the malicious code is stored in its place. Upon being attested, an attacker could generate a valid response from the correct code image and send it back (time-of-check case). The malicious code image replaces the correct code image to be executed (time-of-use case).
- The correct code image is located in its correct location in program flash memory so that the attestation will succeed (time-of-check case). However, the malicious code is executed from another location of program flash memory (time-of-use case).
- Neither the correct image nor malicious image is in the right location of program flash memory. The malicious image is executed whereas the correct image is attested. Obviously, this case is a combination of the above two cases.

In our scheme, the code image to be attested is hashed and stored in the non-volatile area of the TPM at the bootloader stage (line 2 in Algorithm 2). This hashed value is read from non-volatile area of TPM at the application stage for attestation ($\{V_{PCR_1}, N_B, Digest_B\} \leftarrow TPM\_NVReadValue(NV\_Area1, N_A)$ in Figure 3.3). In order to use the first case of TOCTTOU attack, the attacker could take either one of the following two actions:

- The attacker returns the correct attestation response (V_{PCR_1} in Figure 3.3) rather than retrieve the one from the non-volatile area of the TPM at the *application stage*. However, the attacker could not simply return the correct attestation response (i.e., V_{PCR_1} in Figure 3.2) since the response nonce (N_B) and the proper authorization digest ($Digest_B$) are also required (see Figure 3.2). The output nonce returned to the challenger would be different in each attestation process due to the randomness of the input nonce (N_A in Figure 3.2). The signature is over the response nonce ($\{V_{PCR_1}, N_B, Digest_B\} \leftarrow TPM\_NVReadValue(NV\_Area1, N_A)$ in Figure 3.3) so it will be different despite the same application code. The attacker could compromise node B but could not get access to the $Key_{B_{private}}$ to cover the response nonce given that $Key_{B_{private}}$ is stored in the TPM.
- The attacker injects malicious application code over-the-air. The malicious application code could attempt to write the correct response value (V_{PCR_1} in Figure 3.3) into the non-volatile area at the application stage before the wrong attestation response value is read from the non-volatile

area ($\{V_{PCR_1}, N_B, Digest_B\} \leftarrow TPM\_NVReadValue(NV\_Area1, N_A)$ in Figure 3.3). However, the shared secret between the node and base station (K_{BS} for node B) varies from node to node. The correct attestation response for each node is different despite the same application code because this shared secret is incorporated in the attestation response (line 2, 6 in Algorithm 2). The attacker could learn the correct response (V_{PCR_1} in Figure 3.3) through physical node compromise. However, it could not discover the correct response value for other nodes through over-the-air code injection. As we argued in Section 2.3, the physical node compromise is impractical for large-scale WSN as it would require the attacker's device to be present in the field.

In order to apply the second case of the TOCTTOU attack, the attacker needs to make the program counter jump to the address where the malicious code is stored. The attacker could change the application code to implement this, but this will eventually be detected by the attestation process. Another alternative is to invoke the existing routines in a different execution order to form a malicious code. Mal-packet attack [7] is one such attack. After discovering a software vulnerability, the attacker injects a dedicated *mal-packet* that exploits a buffer overflow to execute the existing instructions stored in the program flash in a different sequence, performing some tasks such as overwriting the sensing data. After that, packet propagation instructions will be invoked to transmit the mal-packets to other nodes. Such an attack might not need to change the application code itself, but it would require exploiting the routine that processes the received broadcast packets for execution sequence changes and self-propagation. The packet-processing routine could be included as part of the application code in our scheme. Exploitation of the packet-processing routine could lead to a change in the attestation response, resulting in attestation failure.

The third case of the TOCTTOU attack is a combination of the previous two attacks, and so it could be defended against as long as either one of them is detected.

Although the attacker could reset the TPM modules (i.e., compromise the root of trust) by capturing the sensor nodes physically, it is not scalable for attacker to physically capture a large number of sensor nodes. In addition, physically capturing large number of sensor nodes requires the attacker's device being present in WSN for along time, which increases the chance of an attacker being detected.

4.2 Rootkit-based Attack

In the rootkit-based attack, two Return-Oriented Programs (ROPs) are required: a program memory hook and a data memory hook [1]. The program memory hook is triggered when the attestation request is received. It copies itself (no more than 700 bytes) to the unused data memory, overwrites the return address of attestation process in the stack to point to the data memory hook, and transfers the malware to EEPROM. Since the malware together with two ROPs are removed from the program flash memory, the attestation response would pass. After the attestation process, the data memory hook is activated to restore the malware program memory hook back to the program flash since

it is pointed to by the return address of the attestation process. In our protocol, the temporary removal of the malware at application stage could not yield the correct attestation response since when a new application image is installed, the bootloader is invoked and the attestation response is written into the non-volatile area of TPM (line 6,7 in Algorithm 2). If the temporary removal of the malware occurs at the bootloader stage rather than at the application stage, it will require bootloader compromise and is not scalable as the bootloader can only be changed through use of a hardware programmer.

4.3 Cuckoo Attack

Parno et al. proposed *cuckoo attack* to hijack the communication between TPM and the TPM-enabled PC [15]. In this attack, the malware on the local host proxies all the messages to/from TPMs during the remote attestation to a remote, TPM-enabled machine controlled by the attacker. The attacker's TPM_M can produce an Endorsement Certificate to certify its own public key ($Key_{M_{public}}$) comes from an authentic TPM. At the same time, TPM_M needs to participate in the attestation protocol, providing a correct attestation response regardless of the actual state of the hijacked host. In the context of our remote attestation protocol, the attacker will not only need to redirect the traffic between attestation responder and its TPM to the attacker's TPM (i.e., the TPM commands within attestation responder B shown in Figure 3.2), but will also be required to hijack the traffic between challenger and the sink/trusted third party since the challenger retrieves the public key of node B ($\text{Key}_{B_{public}}$ in Figure 3.2) from the sink rather than from TPM board installed in attestation responder, described in [15]. In order to hijack the traffic between challenger and the sink, the attacker needs to crack the shared secret between node A and node B (K_{AB} in Figure 3.2) to forge the responses from the sink to challenger (i.e., replacing $E_{K_{AB}}(Key_{B_{public}}||V_{PCR_1})$ with $E_{K_{AB}}(Key_{M_{public}}||V'_{PCR_1})$ in Figure 3.2). There are two ways to crack K_{AB}. One way is to physically compromise either node A or node B during their attestation process (see Figure 3.2). The physical node compromise requires the attacker's device being present in WSN for a long time, which increases the chance of attacker being detected and is not scalable for larger number of sensor nodes. Moreover, the limited access to K_{AB} makes it difficult on attackers to retrieve K_{AB} despite the node compromise (i.e., K_{AB} is directly available after K_{AB} is unsealed and before K_{AB} is removed in Figure 3.2. In other time, K_{AB} is sealed and difficult to crack.) The other way is to brute-force K_{AB} through the packet interception. However, the number of packets encrypted by K_{AB} is limited in our protocol (Challenge: $E_{K_{AB}}(N_A)$ and $E_{K_{AB}}(Key_{B_{public}}||V_{PCR_1})$ in Figure 3.2) and the length of K_{AB} is 20 bytes [14], which requires 2^{159} flip-flop operations (on average) to brute-force K_{AB}. The effort for the attackers is daunting and is not scalable for large sensor networks[1].

[1]Given a 16-byte key, a device that can check 10^{18} keys per second needs 10^{13} year to exhaust the complete key space, which is longer than the age of universe.

4.4 TPM Reset Attack

When the value of PCR is changed, it is extended rather than overwritten by the new value (i.e., $V_{PCR_{new}} = H(V_{PCR_{old}}||V_{new})$ rather than $V_{PCR_{new}} = H(V_{new})$). Therefore, the attacker cannot simply place the correct new value in the PCR through $TPM\_PCREXTEND$ command to forge a correct attestation response. However, Lawson et al. recently proposed to reset the stored value in PCR through restarting the TPM [12]. In the context of our attestation protocol, the attacker could simply input a correct hashed value after he/she resets the PCR old value by restarting the TPM *at application stage* (i.e., At application stage, the attacker resets $V_{PCR_{old}}$ to default value by sending $TPM\_Turnoff$ and $TPM\_Startup$ commands, then invokes $TPM\_PCREXTEND$ from line 6 in Algorithm 2 to reconstruct the correct TPM responses.). According to TPM specification, there are two different modes for $TPM\_Startup$[14]. One is *clear mode*, which will clear all history information stored in PCRs after TPM is switched off. The other one is *set mode*, which will preserve the history information stored in PCRs after TPM is off. At application stage, though $TPM\_Startup$ is executed in set mode, the attacker can construct the clear-mode $TPM\_Startup$ by alternating the corresponding flag value given the set-mode $TPM\_Startup$. In the PC platform, a TPM is always turned on and this attack could not be launched without restarting the host, which extends the current platform configuration into the reset PCR through BIOS, therefore making the attacker's effort futile (i.e., the reset $V_{PCR_{old}}$ is extended by BIOS when the TPM-enabled PC is boot up, before the attacker forged a correct extended value). However, in the sensor platform, keeping TPM always activated is not affordable given the limited power supply in WSNs (see Table 5.3, 5.4). Therefore, we proposed to store the PCR extended values in the non-volatile storage area of TPM (i.e., line 5,7 in Algorithm 2). These non-volatile storage areas are not subject to the startup mode of a TPM and are secured by the shared secret (K_{BS} in Algorithm 2) only known to the bootloader. This shared secret is unique for each node so that by physically capturing a few nodes, the attacker could not learn about these secrets for other nodes. Hence, the TPM reset is not able to remove the history values of PCRs stored in these non-volatile areas by restarting the TPM without resetting the micro-controllers.

4.5 Man-in-middle Attack

In order to attest the sensor node multi-hop away, we adopted an approach similar to the expanding ring in SCUBA [18]. The challenger first attests its one-hop neighbors as described in Section 3.2. Then it will choose the verified nodes as surrogates to further attest the nodes two-hop away and report back the attestation responses. This attestation process is iteratively expanded to nodes multi-hop away. One of the potential threats in this method is man-in-middle attack. The attacker might physically compromise the intermediate nodes after these nodes pass the attestation of their previous hops and before they attest their next hops. Although the compromised nodes can not forge a legitimate attestation responses for the malicious nodes at application stage since the attestation responses are generated at bootloader stage (line 7 in Algorithm 2), they can frame the downstream healthy nodes by tampering with their correct attestation responses back to sink. This framing attack can succeed only if the

set of colluding compromised nodes could completely isolate the framed sensor nodes from other healthy peers. As long as one sensor node reports a correct attestation response for the framed sensor nodes, the challenger will attest the sensor nodes reporting the different attestation response again, therefore discovering the compromised intermediate sensor nodes. The corresponding latency and memory overhead of this method will be evaluated in Section 5.5.

4.6 Implementation Issues

In Harvard architecture, the program flash is separated from the data memory. Since the data memory is not directly executable and the attacker needs the instructions from Bootloader (e.g., SPM instructions) to copy the injected code image into the program flash [6], the attestation protocol could simply verify the entire program flash, whose content is known to the sink a priori. Therefore, it is feasible to verify a run-time sensor node if its microcontroller adopts the Harvard architecture. On the other hand, in Von Neumann architecture, the program image and code data share the same memory address space. Since we want to verify the code image but could not externally determine how much memory would be used for the image. Therefore, we need to verify the entire memory space, which contains the dynamic components in run-time such as the program stack pointer. The challenger will need to know the exact execution state of the Von Neumann-based sensor node when it is verified. Therefore, the attestation process in Von Neumann architecture is slightly different. Several checkpoints are preset in the code image which all dynamic state present in memory, with the possible exception of environmentally-influenced state like sensor readings, is externally predictable. If the challenger could verify the code image at these checkpoints, the runtime code image is still verifiable in Von Neumann architecture.

5 Performance Evaluation

5.1 Communication Overhead

In our design, for the challenge (i.e., $E_{K_{AB}}(N_A)$ in Figure 3.2), the payload of the nonce (N_A) is 20 bytes[1] due to the specification of the input nonce in Trusted Computing standard[14]. Therefore, it could be accommodated into one packet given that the maximum packet payload in 802.15.4 standard is 102 bytes. For the response (i.e., Signature, $E_{K_{AB}}(N_B||Digest_B)$ in Figure 3.2), the total size of the response is 296 bytes, where the signature size is 256 bytes [17] and the size of N_B and $Digest_B$ is 20 bytes each [14, 4]. Therefore, at least three packets are theoretically required to transmit the attestation response back to the challenger. In our design, payload size of each data packet is 20 bytes since some of packet payload is reserved for the packet headers in the lower layer (e.g., Medium Access Control (MAC)) and N_B and $Digest_B$ can be placed into different data packets so that the challenger does not need to parse them from a bulky packet. Therefore, the number of packets for the attestation response in our remote attestation protocol is 15.

[1]The size of $E_{K_{AB}}(N_A)$ is the same as N_A since it is the symmetric encryption.

Table 5.1: The response latency in attestation process (single responder)

attestation phases	**latency(s)**
challenge generation	5
challenge transmission and response receipt	8
response verification	9
the entire attestation process in Figure 3.2	22

5.2 Response Latency

During the attestation process at the application stage, there are three main phases.

- Challenge generation (from $N_A \leftarrow TPM\_Random$ to Challenge:$E_{K_{AB}}(N_A)$ in Figure 3.2)
- Challenge transmission and response receipt (from Challenge: $E_{K_{AB}}(N_A)$ to Response:$Signature$, $E_{K_{AB}}(N_B||Digest_B)$ in Figure 3.2)
- Response verification (from Response:$Signature$, $E_{K_{AB}}(N_B||Digest_B)$ to $TPM\_VerifySignature$ $(Key_{B_{public}}, V_{PCR-1}||N_B, Signature)$ in Figure 3.2)

Table 5.1 shows the response latency measured in each phase during the attestation process. The entire attestation process takes less than half a minute. More than 60% latency is incurred at the challenger. Such a result is expected as the challenger could be the sink or trusted third party, which has sufficient power and resource to perform the TPM commands in challenger side. As to the attestation responder, it takes only 8 seconds to complete the response, including the network propagation delay, which shows that TPM-enabled attestation protocol is efficient in terms of latency. In addition, the low latency in attestation responder implies the efficient power consumption in sensor node, which is analyzed in Section 5.4.

5.3 Memory Overhead

The memory overhead in our attestation protocol (illustrated in Figure 3.2) is listed in Table 5.2. After the TPM library and radio driver is loaded, the RAM usage increases from 0.48 kB up to 4.5 kB while the ROM usage ascends from 24kB to 68kB. The memory usage in this evaluation can be further optimized and we believe that it can be further reduced. How to optimize the memory usage of sensor nodes is beyond the scope of this paper.

5.4 Power Consumption

The power consumption of the attestation protocol is attributed to the following three aspects:

Table 5.2: Memory overhead comparison.

Applications	**ROM (kB)**	**RAM (kB)**
Blink (No attestation)	24	0.48
Challenger	68.03	4.51
Attestation responder	68.11	4.50

- The executions of TPM commands;
- The challenge and response transmission/reception (Challenge:$E_{K_{AB}}(N_A)$ and Response:$Signature$, $E_{K_{AB}}(N_B||Digest_B)$ in Figure 3.2);
- The symmetric key operations ($E_{K_{AB}}(N_A)$ and $E_{K_{AB}}(N_B||Digest_B)$ in Figure 3.2).

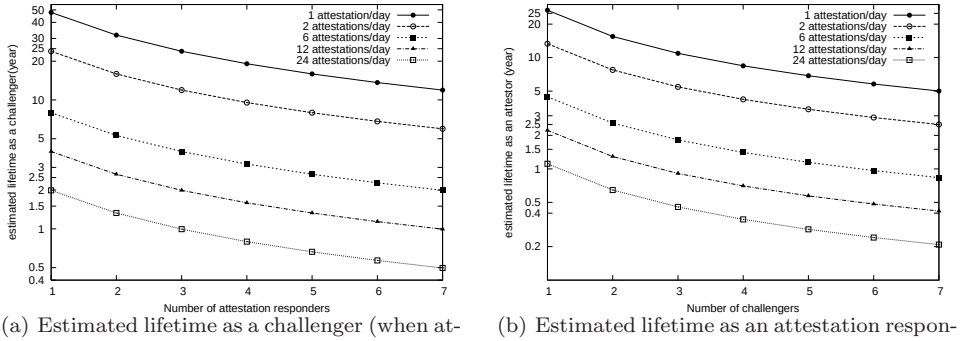
The power consumption of each item in challenger is listed in Table 5.3, where some of the measured data are from secFleck. In our attestation protocol, TPM is switched off by the sensor nodes until the sensor nodes receives the challenges. Therefore, the major power consumption in our attestation protocol is attribute to the TPM commands. For the challenger, most power consumption is due to $TPM\_Unseal(K_{AB}, PCR_2)$, which incurs 70$\mu$J (hightlighted in Table 5.3). The power consumption for other operations is negligible compared with this TPM command. As to the attestation responder, TPM_Sign incurs the most power consumption (highlighted in Table 5.4). This phenomenon might lead to the power depletion attack as the attacker could simply send the excessive challenges to request the attestation response, which would be part of our future works.

In order to evaluate the impact of our attestation protocol on the sensor node lifetime, we make the following assumptions:

- The node is powered by 2 AA 2800 mAHr batteries, which yields $2,800 * 2 * 1.5 * 3,600 = 3.024 * 10^7 mJ$.
- We computed the estimated lifetime of a sensor node based on the power consumptions listed in Table 5.3 and 5.4.
- 10% of the battary power is used for remote attestation only.

The estimated lifetime of a sensor node is illustrated in Figure 5.1 if this node is attested by multiple nodes as an attestation responder (Figure 5.1(a)) or if this node attests multiple nodes as a challenger (Figure 5.1(b)) [2]. This evaluation results quantify the impact of our remote attestation protocol on reasonable lifetime of sensor nodes. For example, the lifetime of a sensor node can last for 6 years if it challenges 7 nodes, twice per day for each node (Figure 5.1(a)). On the other hand, if a node is attested 12 times per day, the lifetime of this node will be 0.4 to 2.2 years, depending on number of challengers (Figure 5.1(b)).

[2]Both tests are done in sequential mode for single-hop network, which is explained in Section 5.5.



(a) Estimated lifetime as a challenger (when attesting multiple nodes).

(b) Estimated lifetime as an attestation responder (when being attested by multiple nodes).

Figure 5.1: The lifetime estimation for a sensor node running our attestation protocol.

5.5 Scalability Test

In order to evaluate the scalability of our remote attestation protocol, we assume that the sink (base station) wishes to attest *all* the nodes in WSNs. We deployed up to seven nodes as attestation responders in our tests. All the nodes in the multi-hop topologies are divided into different *hop groups* based on their hop distances to the sink (e.g., in Figure 5.2(a), node 3 and 4 are in the same hop group). We implemented two basic attestation modes for the attestation for single-hop network and multi-hop network:

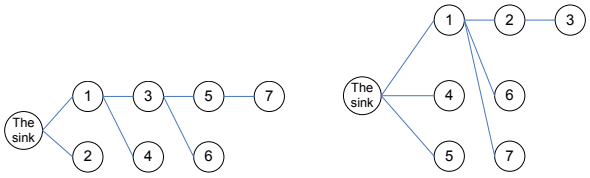
- *concurrent mode*: for the single-hop network, the challenger node broadcasts one challenge to multiple sensor nodes simultaneously and verifies the responses sequentially. For multi-hop networks, it means the nodes in different hop groups are attested in parallel.
- *sequential mode*: for the single-hop network, the challenger will not attest the next sensor node until the response verification for the previous one has finished. For multi-hop network, any node will not attest its downstream neighbors until it has been attested.

We deployed up to seven nodes as attestation responders in our tests. The simplest topology for the concurrent mode is that all the attestation responders are within one-hop transmission range of the sink (i.e., 1 hop, maximum 7 nodes/hop in Figure 5.3). Similarly, the simplest topology for the sequential mode is that the attestation responders are placed in a straight line, with the first attestation responders directly attested by the sink. The remaining 6 nodes act as one-hop neighbors with respect to their previous peer (i.e., maximum 7 hops, 1 node/hop in Figure 5.3). Any multi-hop topology is a combination of the above two topologies. Therefore, we evaluated two additional topologies:

- *Maximum 4 hops, 2 nodes/hop:* the maximum number of children each node can have is 2, and the maximum hop count is 4 (see Figure 5.2(a)).

Table 5.3: The power consumption in challenger: the operation incurring the most power is highlighted.

Operations	**Current(mA)**	**Execution Time(ms)**	**Energy (μJ)**
$TPM\_GetRandom$	2	1	0.33
$TPM\_Unseal(K_{AB}, PCR_2)$	46	510	70.33
$E_{K_{AB}}(N_A)$(encryption)	8	18	0.43
Transmission of $E_{K_{AB}}(N_A)$ (20 bytes)	36.8	3.68	0.81
Reception of $Signature$(256 bytes), $E_{K_{AB}}(N_B\|\|Digest_B)$(40 bytes)	18.4	3.68	3.04
$E_{K_{AB}}(N_B\|\|Digest_B)$(decryption)	8	36	0.86
$Digest_B$ verification	51.2	13.28	2.04
$TPM\_VerifySignature$	50.4	59	8.91



(a) Maximum 4 hops, 2 nodes/hop .

(b) Maximum 3 hops, 3 nodes/hop .

Figure 5.2: Additional topologies for the scalability test: the numbers indicate the order they are added when the network grows.

- *3 hops, maximum 3 nodes/hop:* the maximum number of children each node can have is 3, and the maximum hop count is 3 (see Figure 5.2(b)).

Depending on the attestation mode for the nodes within the same hop group and that for the different hop groups, there are four possible combined attestation modes for the above two additional topologies given two basic attestation modes (concurrent/sequential mode).

- *Mode A (concurrent concurrent mode)*, where the nodes in the same hop group are attested in parallel and multiple hop groups are attested concurrently as well.
- *Mode B (concurrent sequential mode)*, where the nodes in the same hop group are attested sequentially while multiple hop groups are attested at the same time.
- *Mode C (sequential concurrent mode)*, where the nodes in the same hop

Table 5.4: Power consumption in responder: the operation incurring the most power is highlighted.

operations	**Current(mA)**	**Execution Time(ms)**	**Energy (μJ)**
(Bootloader stage)PCR_Extend	51.2	13.28	2.04
(Bootloader stage)NV_Write	10	11	0.33
Reception of $E_{K_{AB}}(N_A)$ (20 bytes)	18.4	3.68	0.41
$TPM\_Unseal(K_{AB}, PCR_2)$	46	510	70.32
$E_{K_{AB}}(N_A)$(decryption)	8	18	0.43
NV_Read	2	2	0.012
$E_{K_{AB}}(N_B \| \| Digest_B)$(encryption)	8	36	0.86
TPM_Sign	60.8	787	143.44
Transmission of $Signature$ (256 bytes)			
$E_{K_{AB}}(N_B \| \| Digest_B)$ (40 bytes)	36.8	3.68	6.09

group are attested simultaneously while the hop groups are attested one by one according to their hop distances to the sink.

- *Mode D (sequential sequential mode)*, where the nodes in the same hop group are attested sequentially while the hop groups are attested one by one according to their hop distances to the sink.

Take the topology in Figure 5.2(a) as example, the sink can attest node 1 and 2 while node 3 can attest node 5 and 6 at the same time in mode A/B. There are two problems for mode A/B. The first problem of mode A/B is attributed to network security. For example, in Figure 5.2(a), the sink cannot trust the attestation results of node 5 and 6 from node 3 since this attestation is processed before node 3 is attested. Node 3 itself cannot be trusted, let alone the attestation results from it. The same security issue also exists in the mode where the sink attests the nodes multi-hop away only with some intermediate nodes routing the attestation packets between them. Take the topology in Figure 5.2(a) as example, the sink can send the challenge to node 3, relayed by node 1. Upon receiving the challenge, node 3 returns the attestation response back to sink, routed by node 1. However, if node 1 is compromised, it can launch the man-in-middle attack to interrupt the entire attestation process. Therefore, attesting the intermediate nodes before they participated into the attestation process is important in multi-hop attestation protocol, which indicates that mode C/D is better than mode A/B in terms of network security.

The second problem is related to efficient and safe share resource (TPM) in each node. For example, in Figure 5.2(a), due to the share TPM resource between the attestation challenger and the attestation responder logic, it is difficult for node 1 to attest node 3 and 4 while node 1 itself is being attested

by the sink. However, node 3 does not have this issue since it is not attested by any other nodes when node 1 and 2 are being attested. Therefore, mode A/B is difficult to maintain in large-scale networks when the topology becomes much more complicated.

For mode C/D, the sink needs to attest its one-hop neighbors first concurrently/sequentially. Once the one-hop neighbors of the sink has passed the attestation, these verified node can sequentially attest their one-hop downstream neighbors in concurrent/sequential mode. In these two modes, each node is always attested before it attests others. Therefore, the network security and the resource-sharing problems in mode A/B can be avoided. Although the man-in-middle attack is still possible after a node is attested and before it attests others. The available period for this attack is limited and this attack can be detected by the sink, as described in Section 4.5.

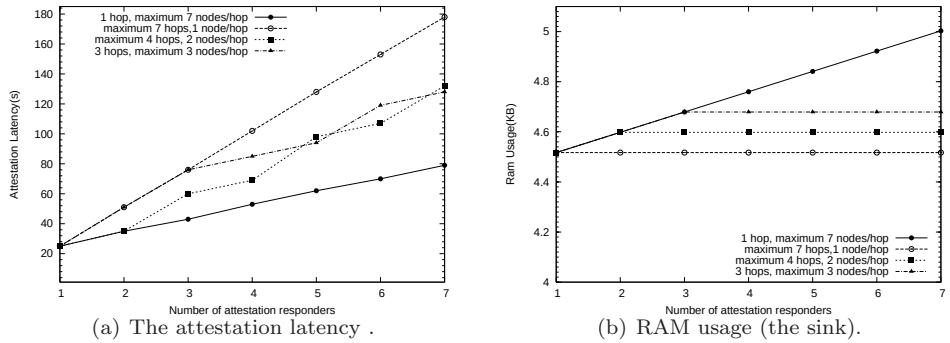
The performance of the mode D is always the same as the simplest topology for the sequential mode since one node is always attested after another regardless to the topology (i.e., maximum 7 hops, 1 node/hop). Therefore, we adopt mode C for these two additional topologies to study the impact of this design choice on our scalability test.

The attestation latencies for these four topologies are shown in Figure 5.3(a). The response latency in 7 nodes/hop grows much slower than that in 1 node/hop as the number of nodes to be attested increases. The attestation latency for 2 nodes/hop topology ascends more significantly each time every two nodes are added into the network since the number of hops grows. On the other hand, the attestation latency for 3 nodes/hop topology is the same as 1 node/hop topology when network size is not larger than 3 nodes. After that, it follows the same incremental rate as the 7 nodes/hop topology with every three nodes added because every three additional sensor nodes, including the first node in each hop group, are attested in parallel.

Although mode C prevails mode D with respect to attestation latency, mode C requires the challenger to set aside multiple buffers to store the attestation responses from multiple nodes, which incurs more memory overhead than mode D[3]. Therefore, in Figure 5.3(b), the RAM usage of the challenger application in the topology of 1 hop, maximum 7 nodes/hop linearly grows as the number of sensor nodes to be attested increases. On the other hand, the RAM usage of challenger application in the topology of 7 hop, 1 node/hop is independent of the number of nodes to be attested since the sink simply challenges its one-hop neighbors, which subsequently attest other downstream peers. The memory overhead of 2 nodes/hop and 3 nodes/hop also increases linearly as the actual number of attestation responders in each hop grows until it reaches the maximum number of nodes in each hop. Therefore, the scalability test results show that our remote attestation protocol is affordable even in the multi-hop remote attestation.

Based on the comparison in Figure 5.3, there is trade-off between mode C and mode D in terms of the latency and memory overhead. If the challenger is the sink or base station, mode C is better given the unlimited resource compared to the sensor nodes. On the other hand, if the sensor network is dense (i.e., the average number of one-hop neighbors is large) and the challenger is a

[3]For the responders, since the attestation responses are processed individually, the memory overhead is the same as single-node attestation.



(a) The attestation latency . (b) RAM usage (the sink).

Figure 5.3: The scalability test result for topologies listed in Section 5.5.

sensor node, limited in memory, mode D is better since the memory overhead is independent in sequential mode.

6 Related Work

6.1 New attacks and the preventative schemes

Although worm attacks in the Internet [22] have been investigated extensively, work on such attacks in WSNs, particularly for Harvard-architecture nodes, remains at an early stage. Francillon et al. proposed *meta-gadgets* to inject the malware into data memory, copy it from the external flash memory to RAM, then duplicate the malware from RAM to program memory using the dedicated SPM instruction from the bootloader [6]. Such an attack could be catastrophic once the malware self-propagates by calling the packet transmission function. The authors in [6] provided a detail discussion of this attack but have not provided any comprehensive solutions. Yang, et. al. proposed a software diversity approach to improve sensor network immunity against sensor node worm attack [26]. Their method requires grouping the sensor nodes so that the nodes from neighboring groups do not have the same version of the application. Consequently, the attacker needs to exploit all the available versions of the application code to compromise the whole network, which increases the difficulty of the attack. However, grouping the nodes is an expensive operation (in fact, an NP-complete problem) and requires a centralized entity (e.g., base station) to implement it. In addition, the software diversity approach is not scalable in a dense WSN as some basic routines (e.g., sending out a packet) could not possibly have many fundamentally distinct alternative implementations. Gu et al. proposed a mal-packet attack to create malicious code by altering the execution flow of the existing routines in the program memory [7]. This attack requires fewer operations as the attack does not require injection of new code into the target node. It is also harder to detect as it does not change the actual content of the program memory. However, such an attack has several limitations as it only exploits existing routines and the mal-packet attack disappears when the nodes are reset. Later, Ferguson et al. proposed a self-healing scheme to resume

the legitimate execution sequence of control flow in the sensor node when the sequence is altered by mal-packets [5]. Basically, the application is separated into Atomic Code Blocks (ACBs). Each ACB is assigned a unique marker. The execution order of these unique markers is monitored. If the execution order is changed, the current executing ACB is terminated and the next ACB in the original executing order is invoked. Their approach assumes that the original execution order of ACBs cannot be compromised. Hence, protection of the original execution order of ACBs from tampering could be another significant security issue. Such security issues are one of our motivations for introducing the TPM for enhancing WSN security: it provides a hardware-based *root of trust* stored in TPM, to be further discussed in Section 3.1.

6.2 Software-based attestation schemes

The work outlined in Section 6.1 concentrates on preventative measures against sensor node worm attack. Measures are also required to detect attacks. The most straightforward detection measure is *remote attestation*, whose aim is to verify the program flash memory of the sensor nodes. Recovery measures are taken if an unauthorized alteration in the program code has been discovered (e.g., exclusion or reprogramming of the corresponding tampered sensor nodes). It is widely believed that sensor nodes cannot afford the additional hardware required to perform the attestation process, so most of the remote attestations procedures proposed for WSNs are software-based. Seshardri et al. presented SWATT (SoftWare-based ATTestation for Embedded Devices) to verify the content of the program memory even while the sensor nodes are running [19]. In SWATT, in addition to a random Message Authentication Code (MAC) key, the verifier requires a MAC over a random segment of the program flash memory as well. In order to circumvent attestation, an attacker would need to interpret the challenge and generate a response from the corresponding segment of the untampered program flash code, which slows the attestation process. The verifier sets a timer right after it sends out the challenge. If there is a timeout before a response is received, the verifier would suspect that the sensor node to be attested has been compromised.

Shaneck et al. further extended SWATT in their paper [21]. In their protocol, the verifier constructs a new attestation procedure for each verification request and sends the attestation code to the sensor node being verified. The attestation procedure uses various code obfuscation techniques such as opaque predicates to make it hard for the attacker to perform static or dynamic analysis of the attestation procedure within the time alloted to the sensor node by the base station for computing the attestation response. Different from SWATT, the network delay is taken into consideration when the verifier computes the expected attestation time. However, the paper does not present any results to substantiate their claim.

SCUBA (Secure Code Update By Attestation in Sensor Networks) is an approach to detect and repair compromised sensor nodes through remote attestation [18]. SCUBA is based on a primitive operation called ICE (Indisputable Code Execution) to dynamically establish a trusted code base on a remote, untrusted sensor node. The verification code in SCUBA is a *self-checksum code*. The self-checksum code is a sequence of instructions that compute a checksum over themselves in a way that the checksum would be either wrong or slower

to execute if the sequence of instructions is altered. As for its predecessors, SCUBA relies on two criteria to determine whether a sensor node being verified is compromised. One is the correctness of the self-checksum response. The other is the response delay. If either one of them does not meet the expectation of the verifier, the verifier will presume the sensor node has been compromised. Such a sensor node would be either repaired through a code update or revoked as compromised node.

Obviously, all the above software-based remote attestation protocols depend on the response time to determine whether an attacker has interfered with the attestation process. However, while performing remote attestation over the network, the network communication or execution state of the sensor node can always introduce some unpredicted delay, resulting in an inaccurate measurement of the response time of the attestation process, and consequent false positives.

Castelluccia et al. presented two new attacks to circumvent malware detection of the above software-based attestation protocols: *rootkit-based attack* against response-time-based attestation [1]. In the rootkit-based attack, the attacker could copy the malware into EEPROM before the attestation starts and restore it in the program flash after the attestation. Despite the incurred response delay due to the malware transfer, the authors argued that their rootkit-based attack incurs additional 7.4% response delay, which is faster than the expected value by SWATT (13%) and their attack could therefore circumvent SWATT's detection [19]. The rootkit attack further strengthens the conclusion that the software-based attestation protocols are not sufficiently secure to defend against memory-related attacks.

6.3 Hardware-based attestation schemes

Christoph et al. proposed a partial hardware-based attestation protocol to detect a compromised node in cluster-based network [11]. In their paper, the sensor nodes are grouped into clusters and each cluster has cluster head, which is a much more powerful device than sensor nodes and is equipped with a cryptographic hardware, i.e. a Trusted Platform Module (TPM). The sensor nodes within the same cluster can challenge their corresponding cluster head. The attestation response is secured by TPM. In their scheme, only the cluster heads can be verified by attestation. In addition, this paper did not provide any testbed evaluation on their scheme but an analytical model with respect to the power consumption.

7 Conclusion and Future Work

In this work, we presented a hardware-based remote attestation protocol in WSNs with the assistance of Trusted Platform Module (TPM). Instead of employing the TPM on the cluster heads only or adopting the software-based attestation design, the additional hardware, TPM, is equipped with *each* sensor node. Each sensor node could be challenged in regard to its program flash content. To our knowledge, our hardware-based attestation protocol is the first one in WSNs with *each* sensor node equipped with the additional hardware. We discussed the potential attacks against our remote attestation protocol, including those recent attacks (e.g., rootkit-based attack) that can practically beat the

software-based attestation protocols, against which we further investigated the counter measures. The corresponding performance evaluation shows that TPM can improve the efficiency of the attestation with acceptable computational and power overhead. Moreover, the scalability test shows that the response latency of our hardware-based attestation protocol is linear to the number of nodes being attested. The sink could handle multiple attestation response in parallel with acceptable extra overhead. Our future work will include the counter-attack against the energy depletion attack against the TPM operations might post another treat against the remote attestation protocol.

Bibliography

[1] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Soriente Claudio. On the difficulty of software-based attestation of embedded devices. In *CCS '09*. ACM, 2009.

[2] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[3] Prabal K. Dutta, Jonathan W. Hui, David C. Chu, and David E. Culler. Securing the deluge network programming system. In *IPSN '06*, pages 326–333. ACM Press, 2006.

[4] D. Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1). RFC 3174 (Experimental), September 2001.

[5] Christopher Ferguson, Qijun Gu, and Hongchi Shi. Self-healing control flow protection in sensor applications. In *WiSec '09*, pages 213–224. ACM, 2009.

[6] Aurélien Francillon and Claude Castelluccia. Code injection attacks on harvard-architecture devices. In *CCS '08*, pages 15–26. ACM, 2008.

[7] Qijun Gu and Rizwan Noorani. Towards self-propagate mal-packets in sensor networks. In *WiSec '08*, pages 172–182. ACM, 2008.

[8] Carl Hartung, James Balasalle, and Richard Han. Node compromise in sensor networks: The need for secure systems. Technical report, University of Colorado at Boulder, January 2005.

[9] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04*, pages 81–94. ACM Press, 2004.

[10] C. Karlof and D. Wagner. Secure routing in wireless sensor networks: attacks and countermeasures. In *Proceedings of the First IEEE International Workshop on Sensor Network Protocols and Applications*, pages 113–127, 2003.

[11] Christoph Krauß, Frederic Stumpf, and Claudia Eckert. Detecting node compromise in hybrid wireless sensor networks using attestation techniques. In *Security and Privacy in Ad-hoc and Sensor Networks*, pages 203–217, 2007.

[12] Nate Lawson. TPM hardware attacks. http://rdist.root.org/2007/07/16/tpm-hardware-attacks/.

[13] An Liu and Peng Ning. Tinyecc: A configurable library for elliptic curve cryptography in wireless sensor networks. In *IPSN '08*, pages 245–256, 2008.

[14] TCG Specification Architecture Overview. Technical report, Trust Computing Group, August 2007.

[15] Bryan Parno. Bootstrapping trust in a "trusted" platform. In *HOTSEC'08*, pages 1–6, Berkeley, CA, USA, 2008. USENIX Association.

[16] John Regehr, Nathan Cooprider, Will Archer, and Eric Eide. Memory safety and untrusted extensions for tinyos. Technical report, School of Computing, University of Utah, 2006.

[17] R. L. Rivest, A. Shamir, and L. M. Adelman. A method for obtaining digital signatures and public-key crytosystems. Technical report, 1977.

[18] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SCUBA: Secure Code Update By Attestation in sensor networks. In *WiSe '06*, pages 85–94. ACM Press, 2006.

[19] Arvind Seshadri, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. SWAtt: Software-based attestation for embedded devices. In *Proceedings of the IEEE S & P*, 2004.

[20] M. Shaneck, K. Mahadevan, V. Kher, and Y. Kim. Remote software-based attestation for wireless sensors. *Proceedings of the 2nd European Workshop on Security and Privacy in Ad Hoc and Sensor Networks*, 2005.

[21] Mark Shaneck, Karthikeyan Mahadevan, Vishal Kher, and Yongdae Kim. Remote software-based attestation for wireless sensors. In *ESAS '05*, pages 27–41. 2005.

[22] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to own the internet in your spare time. In *SSYM'02*, pages 149–167, Berkeley, CA, USA, 2002. USENIX Association.

[23] Hailun Tan, Sanjay Jha, Diet Ostry, John Zic, and Vijay Sivaraman. Secure multi-hop network programming with multiple one-way key chains. In *WiSec '08*, pages 183–193. ACM, 2008.

[24] Ronald Watro, Derrick Kong, Sue-Fen Cuti, Charles Gardiner, Charles Lynn, and Peter Kruus. Tinypk: securing sensor networks with public key technology. In *SASN '04*, pages 59–64. ACM Press, 2004.

[25] A. D. Wood, J. A. Stankovic, and S. H. Son. Jam: a jammed-area mapping service for sensor networks. In *Proceedings of 24th IEEE Real-Time Systems Symposium*, pages 286–297, 2003.

[26] Yi Yang, Sencun Zhu, and Guohong Cao. Improving sensor network immunity under worm attacks: a software diversity approach. In *MobiHoc '08*, pages 149–158. ACM, 2008.

[27] Yanyong Zhang, Wade Trappe, Zang Li, Manali Joglekar, and Badri Nath. Robust wireless localization: Attacks and defenses. In *Secure Localization and Time Synchronization for Wireless Sensor and Ad Hoc Networks*, pages 137–160, 2007.