Proactive Code Verification Protocol in Wireless Sensor Network*

Young-Geun Choi, Jeonil Kang, and DaeHun Nyang

Information Security Research Laboratory,
INHA University, Republic of Korea
{choizak,dreamx}@seclab.inha.ac.kr, nyang@inha.ac.kr

Abstract. For WSN(Wireless Sensor Network) to provide reliable service, authentication is one of the most important requirements. The authentication usually means the entity authentication, but owing to the data centric nature of sensor network, much more importance must be put on the authentication(or attestation) for code of sensor nodes. The naive approach to the attestation is for the verifier to compare the previously known memory contents of the target node with the actual memory contents in the target node, but it has a significant drawback. In this paper, we show what the drawback is and propose a countermeasure. The basic idea of our countermeasure is not to give the malicious code any memory space to reside by cleaning the target node's memory space where the malicious code can reside. This scheme can verify the whole memory space of the target node and provides extremely low probability of malicious code's concealment without depending on accurate timing information unlike SWATT[1]. We provide this verification method and show the performance estimation in various environments.

1 Introduction

In WSN(Wireless Sensor Network), security is one of the most important issues. Let us assume that WSN is used for military surveillance. If some of nodes are captured by the enemy force and compromised to work abnormally, these nodes cannot detect intrusion of the enemy or can report false information to deceive our forces. To prevent this abnormal behavior of sensor nodes, a method to attest suspicious nodes in WSN is required.

Unfortunately, physical comproming attack cannot be avoided by any means. Only hardware tamper-proof module in the node forces the node not to be modified, and it prevents the node from becoming a malicious node which may attack other nodes. Therefore, researches mainly have been focused on how to detect the compromised nodes remotely. Most of these studies, however, work only after the compromised node makes some misbehavior. Thus, until the nodes make some misbehavior, there are no way to find out those malicious nodes.

* This work was supported by grant No. R01-2006-000-10957-0 from the Basic Research Program of the Korea Science & Engineering Foundation.

O. Gervasi and M. Gavrilova (Eds.): ICCSA 2007, LNCS 4706, Part II, pp. 1085–1096, 2007.

Because of this nature, WSN is prone to get the damage caused by the malicious nodes.

Thus, we need methods to detect compromised nodes proactively before they make any misbehavior. SWATT(SoftWare-based Attestation for embedded device) [1] is designed for this remote software based attestation. SWATT is very simple to be adopted, but it works in time critical manner and thus, it is very sensitive to unpredictable network delay and also dependent from hardware platforms. The method newly suggested by Mark Shaneck is based on SWATT technique, but revised the time-related weakness of SWATT[2].

In this paper, we suggest a totally different proactive code verification algorithm to detect compromised nodes. Our scheme can verify the whole memory space of the target node and provides extremely low probability of malicious code's concealment without depending on accurate timing information unlike SWATT[1]. We provide this verification method and show security analysis and performance estimation in various environments.

2 Assumption, Threat model and Requirements

2.1 Assumption

Before we describe our verification protocol, we touch on some assumptions.

We assume that the verifier has copies of the memory contents of a target node. Each entity is assumed to share a pair-wise key to construct secure communication channel between them. The establishment of pair-wise key can be achieved by random key predistribution schems, etc. The remotely executable verification code by verifier is embedded in memory of a target node. The attacker can obtain whole competence for memory access but it is not supposed that the attacker can modify hardware architecture of captured node because it means the attacker can use external resources. We assume that the programmable flash memory of the target node must not have any empty space unless it is for some special purpose. Also, the node is assumed not to support virtual memory and not be able to execute any codes in the external data storage directly.

2.2 Threat Model

The naive approach to attestation is that a verifier compares the known memory contents of a target node with the actual memory contents in that node. The verifier demands testimony to the target node first. And then, the target node executes the verification code in its memory. The verification code accesses and sends the memory contents to the verifier. Finally, the verifier decides whether the target node is infected or not by matching received memory contents to its copy.

However, the attacker can avoid this naive attestation easily using the following attack. He puts some attack code in the sensor node that has valid infomation about original memory contents. And, he modifies the verification code to jump up to the attack code instead of to firmware code. The malicious verification

code sends valid memory contents which the attack code has. Consequently, the verifier does not correctly attestate the target node compromised by an attacker.

2.3 Notation

In this paper, we use the following notations if there are not any other comments.

$V \| D$	the concatenation of codes V, D.
$\{D_i\}_{i=1}^{m}$	the concatenation from D_1 to D_m
$K_{A,B}$	the shared key between A and B.
$\mathcal{E}_{K_{A,B}}(\mathtt{m})$	the encryption of message $\mathtt{m}$ with key $K_{A,B}$.
$\mathcal{D}_{K_{A,B}}(\mathtt{c})$	the decryption of ciphertext $\mathtt{c}$ with key $K_{A,B}$.

3 Our Proposal

In this section, we propose a new protocol about software-based verification method between a BS and a node. The main idea is not to give the malicious code any memory space to reside by cleaning the target node's memory space where the malicious code can reside whenever a verification request is given.

WSN is usually composed of huge number of sensors and a few base stations. While BS has enough resources to process heavy tasks simultaneously, a sensor node has very limited resources such as small memory size, low battery capacity and small radio range. In this section, we discuss the method in which a base station as a verifier attests sensor nodes whether it works correctly or not.

There are various types of memory in a sensor node. For example, MICA mote made by UC Berkeley has a 4kB static RAM, a 128kB flash program memory and a 4Mbit external flash memory[3]. Most of architectures of sensor nodes are like MICA mote's. We will denote SRAM and Flash of micro controller as Mem^1. Mem^i where $i \geq 2$ denotes flash program memories or external data storage; normally, only one flash programmable memory is built in micro controller of a

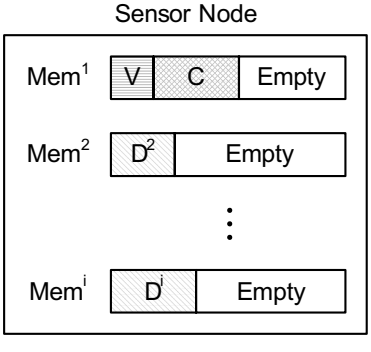
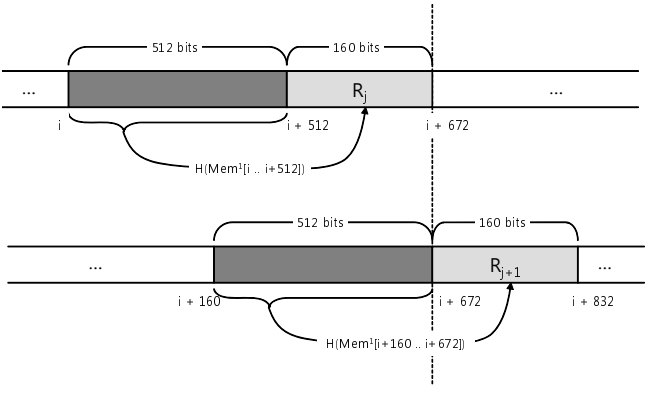
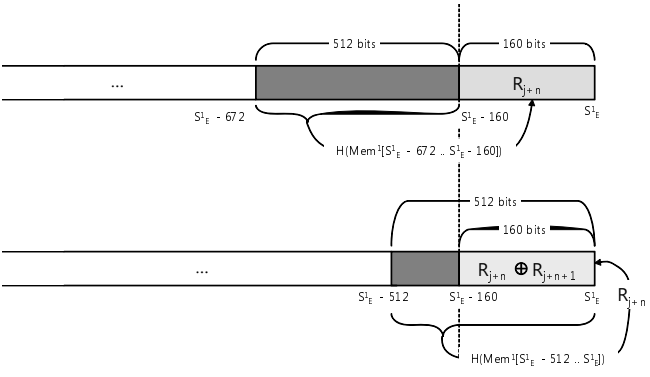


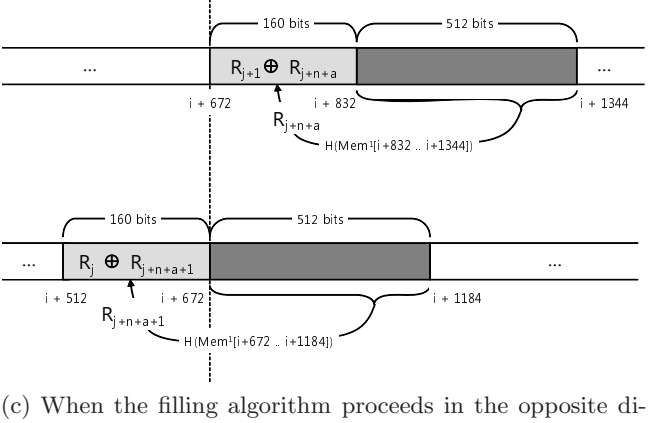
Fig. 1. Memory contents in a sensor node. Mem^1 is a RAM of micro controller and $Mem^i (i > 1)$ is programmable flash memory.



(a) When the random numbers make up the empty memory



(b) When the random number sequence meets end of the empty memory



(c) When the filling algorithm proceeds in the opposite direction

Fig. 2. Filling empty memory space with random number in the operation phase(using SHA-1). It fills the space with the value computed by XOR operation of two random numbers.

sensor node and external data storage is EEPROM. We do not assume any attack using external data storage because the access time of external data storage is too slow to use for the attack and thus it is easily detectable. After sensor nodes are deployed, verification code V and firmware code C are in the Mem^1 as long as it runs. Mem^i has each data part D^i, so all the data in a node is $D = \{D^i\}_{i=1}^{\Lambda}$. Λ denotes all memories in a sensor node. Figure 1 illustrates these memory contents.

Request phase. Our verification mechanism is composed of four phases; the request phase(of the verifier), the operation phase(by the target), the response phase(of the target) and the check phase(by the verifier). Firstly, in the request phase for verifying memory contents of a target node, BS sends the request message (Req), BS's ID (ID_{BS}) and a nonce (n) along with ID_{BS} and n encrypted by shared secure key between the BS and the node ($K_{BS,node}$).

$$\text{BS} \rightarrow \text{node} : Req, ID_{BS}, \mathcal{E}_{K_{BS,node}}(n, ID_{BS})$$

When the node receives this message from the BS, it can check whether these are legitimate or not by comparing ID_{BS} with $\mathcal{E}_{K_{BS,node}}(n, ID_{BS})$. This step can prevent an attacker from impersonating the BS, and thus, the attacker cannot attempt DoS(Denial of Service) attack to unspecific nodes if he does not know the shared key $K_{BS,node}$.

Operation phase. It is the first step of the operation phase that the correspondent node generates random numbers as a seed using n. The empty space should be filled with the bit string made by specific algorithm with random numbers. The size of empty space of Mem^i, S_E^i is the same as the remaining memory space excluding the verification code, the firmware code and the data. It can be calculated by

$$S_E^i = S^i - (S_V^i + S_C^i + S_D^i) \tag{1}$$

The algorithm fills the empty memory space with the bit string computed by XOR operation of two random numbers. Firstly, the random numbers fill the S_E^i. And then, if the random number sequence meets end of the empty memory, it should proceed in the opposite direction and fill the XOR-operated value between existing random number and newly generated random number. Input of hash function for generating random number in this algorithm is previous 512-bits memory contents. Figure 2 shows an example that works following this algorithm. R_j denotes the j-th output of hash function.

However during four rounds of this algorithm the random number can not be generated from previous 512-bit memory contents, so the random number in the first round is made from the seed n. Second to fourth rounds are generated from memory contents and padding 0 value. The reason why the algorithm is used is described in section 4.3. The algorithm Fill_Memory(n) is shown as following.

Algorithm. Fill_Memory(n)

1 **for** i=1 **to** Λ
  /* Filling the empty memory with random number */
2 **for** j=1 **to** 4
3 $n \leftarrow h^{512}(n \parallel \text{padding } 0)$
4 $Mem^i[j-1] \leftarrow n$
5 **for** "j=5 **to** S_E^i
6 $Mem^i[j-1] \leftarrow h^{512}$(previous 512-bit)
  /* XOR operation in the opposite direction */
7 **for** j=1 **to** S_E^i
8 $Mem^i[S_E^i - j] \leftarrow Mem^i[S_E^i - j] \oplus h^{512}$(previous 512-bit)
9 **return**

Response phase. In the response phase, the correspondent node responds to BS with the hash value, $h(\text{MIXED\_MEM}\|D)$ over Mem^1 to Mem^i. MIXED_MEM is the sequence of memory contents that $V\|C\|FM_n$ is read in conformity with the rules.

$$\text{node} \rightarrow \text{BS} : \mathcal{E}_{K_{BS,node}}(h(\text{MIXED\_MEM}\|D), D)$$

The mixing is for preventing that an attacker use a backup from other sensor nodes to hide his code. Figure 3 shows an example.

Since BS does not know sensor node's data D that are temporally stored, usually a node must send D to BS for backup before verification procedure starts. And then, the random number sequence fills all memory space excluding V and C that the BS already knows. After the verification finishes, the node has to receive the D from the BS to restore its original states. This is burdensome

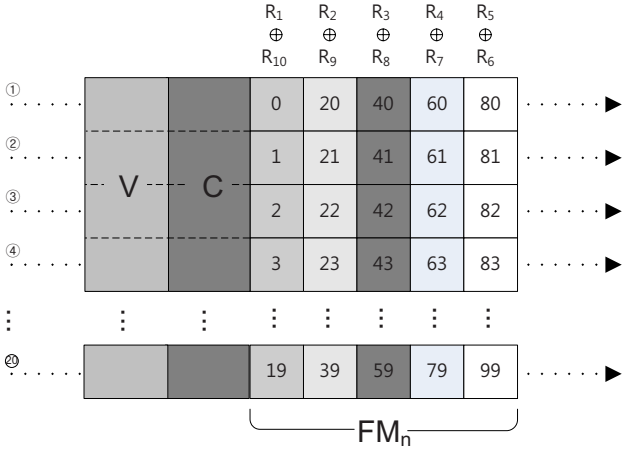
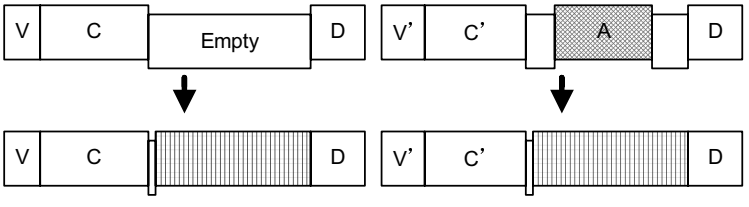


Fig. 3. MIXED_MEM(using SHA-1)

because of two reasons. One is the limited battery capacity of a node. The node consumes energy twice in transceiver mode: sending and receiving. Consequently, a node consumes excessive energy for obtaining D that has been known already before the verification. The other reason is that it increases verification time unnecessarily.

Checking phase. In the proposed protocol, actually a node does not need to receive D. During the verification, a node just sends D with the hash value to the BS. The BS hashes received D with V and C that are already known. And the BS compares the calculated hash value with the received one. If the calculated value equals to the received one, the BS concludes that the node is not compromised. Moreover, when the node does not need to store D because D is only temporary, it just sends $\mathcal{E}_{K_{BS,node}}(h(\text{MIXED\_MEM}))$ to the BS. If the BS knows that the node is compromised or modified illegally, it should notify that of the other nodes.



(a) When a node is normal (b) When a node is compromised

Fig. 4. Change of memory contents of a node before and after running algorithm Fill_Memory()

Figure 4 shows the rough idea of our protocol. That is, if the target is not compromised, it can respond with a correct answer by hashing its whole memory image. If not, even though compromised node reserved an attack code for masquerading, it is overwritten according to Fill_Memory(n). Finally, it is impossible for the node to send correct answer expected by the BS.

4 Security Consideration

4.1 Time Independent Verification

SWATT for WSN works by measuring processing time to detect malicious code. So, if the microprocessor in a node is more powerful, more loops for amplification of small time difference are required. This means that SWATT consumes more energy. Moreover, SWATT for the WSN does not allow unpredictable delay because the delay means existence of malicious code. If allows, an attacker can make a BS recognize code delay as network delay. Therefore, SWATT for the WSN should be adopted very carefully.

On the other hand, our code verification scheme does not need to consider exact processing time. But, if a sensor node has more memory space, it requires more time to fill the empty memory space. However, we can set the threshold time to the value which is long enough to cover processing time and network delay, and this feature of timing independent verification promises more flexible usage because the network delay rarely affects the whole verification time.

4.2 Prevention from Replay Attack

The malicious node which is compromised by an attacker can mount the replay attack against our protocol. The node is able to overhear some messages used in our scheme to send these old messages to the BS which is in verification process with other node. However, our verification scheme is basically a kind of the challenge-and-response protocol, and thus, the replay attack cannot be mounted owing to the nonce, even though the attacker knows the session key between the BS and the other node.

Under the challenge-and-response protocol, if an attacker knows the secret between two entities, he can easily success in the attack by sending the response message which is computed with the secret . Here in our protocol, the secret is the code itself, and so it seems easy to break the protocol because the code is usually known to the attacker. However, it is impossible for the malicious node to make the correct response for a nonce because it does not have enough memory space to generate the message while it keeps up the code both for malicious behavior and for original.

4.3 Prevention of Making Required Random Block on Demand

Another attacking method is that a malicious code generates required blocks of random sequence at the time when the hash function needs the blocks. Let n be the number of block of empty memory, the n-th block can be generated from the seed by $\lceil |S_E^1|/S_{ho} \rceil \times 2 - n$ times hashing because the n-th block is calculated by $R_n \oplus R_{\lceil |S_E^1|/S_{ho} \rceil \times 2 - n}$. where S_{ho} is the output size of hash function, and $|\ \ |$ denotes the size of specific memory. (S_{ho} is 160 if the hash function is SHA-1, and is 128 if MD5.) Remember that the empty memory space is filled twice with random number while the Fill_Memory(n) does. Similarly, $(n-1)$-th block can be generated.

If we assume that the total memory length is very long as compared with the code for attack, the attack code should perform at least 20(SHA-1) or 16(MD5) times of the hash calculation for hiding itself. However the ratio of the attack code in the total memory is higher than our assumption, so that we expect it takes more time.

Actually, there is almost optimal method to compute hash chain using appropriative cache[4]. If using cache, we should consider of the length of memory in each hardware implemented in each sensor node. In this paper, we show examples for proving our protocol's excellence under the assumption that the attack code uses 400 bytes memory(SHA-1) or 320 bytes memory(MD5) (the attack code itself, two 512-bit caches, one 512-bit seed and so on).

5 Analysis of Verification and Attack Model

5.1 Performance Estimation for a Valid Node

Based on our model, we can estimate the total execution time T by

$$T = T_h + \sum_{i=1}^{\Lambda} (T_r + T_a^i) \times S_E^i \tag{2}$$

where T_h is the time for making response, T_r is the time for generating random number and T_a^i is the time for accessing Mem^i.

In most implementations of sensor nodes, only Mem^1, Mem^2 and Mem^3 are used for the sensor, where Mem^1 is SRAM, Mem^2 is programable flash memory, and Mem^3 is the external EEPROM.

Table 1. Hardware flatforms in some applications

Sensor	Microprocessor	Word Size	Clock Freq.	SRAM Size	Flash Size	MIPS
Dot[5]	ATmega163	8-bit	16MHz	1kB	16kB	8
MICA	ATmega128	8-bit	16MHz	4kB	128kB	16
MICA2Dot						
MICA2[6]						
Telos[7]	MSP430F1611	16-bit	8MHz	10kB	48kB	8

If we can say that the flash memory already is filled by an administrator as our assumption, it is enough to fill only SRAM because the writing time in EEPROM is too slow[8]. In some implementations, the size of the external EEPROM is about a few mega bits, and in order to fill that, it will take several thousand seconds. (For example, if the size of EEPROM is 4Mbit and the writing time for 1bit is $1ms$, it takes over 1 hour 8 minutes 16 seconds.) In the other hand, the writing time in SRAM is fast enough to ignore.

As we mentioned before, we could estimate the execution time of our scheme by just counting hash operation, and we can rewrite the equation 2 as

$$T' = T_h + T_r \times S_E^1 \tag{3}$$

Let U_f be the number of hash rounds for filling the empty memory space, and δ be the time to execute one round. Then $T_r \times S_E^1$ can be rewritten by $U_f \times \delta$. Then

$$U_f = \lceil |\text{SRAM}|/S_{ho} \rceil \times 2 \tag{4}$$

Let U_r be the number of hash rounds in the response phase, then

$$U_r = \lceil (|\text{SRAM}| + |\text{Flash}|)/S_{hi} \rceil + 1 \tag{5}$$

Table 2. Round Time and Required Hash Rounds(* means it's an estimated value)

Microprocessor		Round Time (δ)	Rounds (U_f)	Rounds (U_r)
ATmega163	(SHA-1)	* $7272\mu s$	104	272(+1)
	(MD5)	* $2946\mu s$	128	272(+1)
ATmega128	(SHA-1)	$3636\mu s$	410	2112(+1)
	(MD5)	$1473\mu s$	512	2112(+1)
MSP430F1611	(SHA-1)	* $7272\mu s$	1024	928(+1)
	(MD5)	* $2946\mu s$	1280	928(+1)

Table 3. Required Time for Hash Operation

Microprocessor		Time ($U_f \times \delta$)	Time ($U_r \times \delta$)	Total Time (T')
ATmega163	(SHA-1)	$0.756s$	$1.985s$	$2.741s$
	(MD5)	$0.376s$	$0.804s$	$1.180s$
ATmega128	(SHA-1)	$1.490s$	$7.683s$	$9.173s$
	(MD5)	$0.754s$	$3.112s$	$3.866s$
MSP430F1611	(SHA-1)	$7.466s$	$6.756s$	$14.202s$
	(MD5)	$3.770s$	$2.737s$	$6.507s$

where S_{hi} is the input size of one round of hash function. (S_{hi} is 512. +1 is needed because of initial padding operation of hash function.) Then T_h can be rewritten by $U_r \times \delta$. Finally,

$$T' = U_r \times \delta + U_f \times \delta = (U_r + U_f) \times \delta \tag{6}$$

The round times δ in ATmega128, $3636\mu s$ for SHA-1 and 1473 μs for MD5 are shown in Alexander Dean's paper with his experiment[9]. The round times δ in ATmega163 and MSP430F1611 are estimated from ATmega128's one. We calculated the value by comparing the difference of the each MIPS.

Table 3 shows the approximated execution time of our scheme. The estimations are very different in each microprocessor, from 1 second to 14 seconds. The threshold time for waiting response from a node, thus, must be determined considering network delay and these estimated execution time.

5.2 Performance Estimation for an Attacked Node

Now we estimate the optimal attack model that we mentioned in section 4.3. Recall that the attacker have to generates required blocks of random sequence at the time when the hash function needs the blocks. The total execution time T_a by the attacker can be calculated by

$$T_a = T' + U_a \times \delta \tag{7}$$

U_a is the number of hash rounds that the attack code needs to hide itself by generating available response. The size of the attack model we assumed is 20-blocks as we mentioned in section 4.3. Table 4 shows the approximated execution time of the attack model.

Table 4. Required Time for the Attack

Microprocessor		Rounds (U_a)	Time ($U_a \times \delta$)	Total Time for Attack(T_a)
ATmega163	(SHA-1)	3600	26.179s	28.920s
	(MD5)	2880	8.484s	9.664s
ATmega128	(SHA-1)	3600	13.089s	22.262s
	(MD5)	2880	4.242s	8.128s
MSP430F1611	(SHA-1)	3600	26.179s	40.381s
	(MD5)	2880	8.484s	14.991s

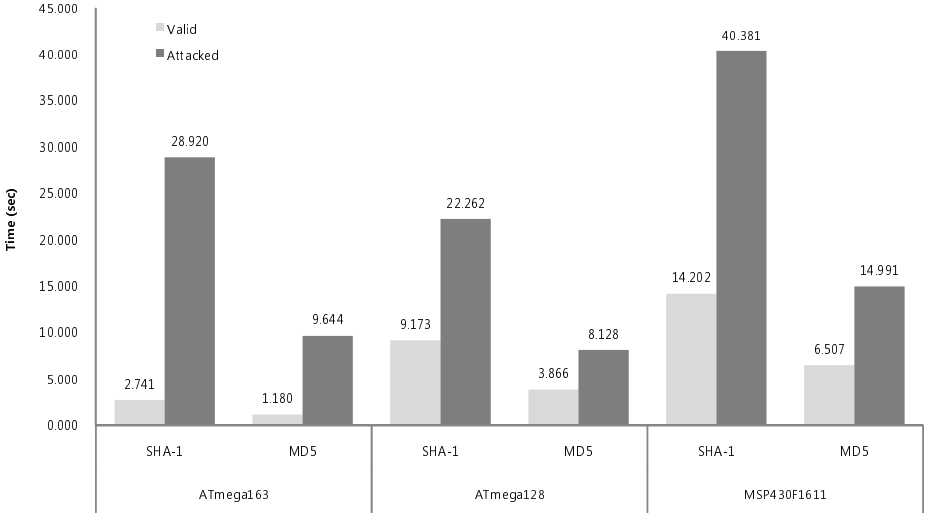


Fig. 5. The comparison between total verification time of valid nodes and malicious nodes. The time of malicious nodes at least doubles that of valid nodes.

Figure 5 shows a gulf between the total execution time and the total attack time. As the figure shows, the total attack time at least doubles the total execution time of valid node. It is enough for the verifier to distinguish the malicious node from valid node.

6 Future Works and Conclusion

In this paper, we suggested a new code verification mechanism. Our scheme works in a very simple way compared with previously suggested methods, but it is good enough to detect malicious codes in sensor nodes without requiring exact timing information.

We can extend our code verification technique to node-to-node mode. Without intervening of base station, each node can cooperatively verify each other's code to banish compromised nodes. Also, proper modification of the protocol can

define a group code verification protocol that can verify codes of group of nodes in efficient manner.

References

1. Seshadri, A., Perrig, A., van Doorn, L., Khosla, P.: Swatt: software-based attestation for embedded devices, pp. 272–282 (2004)
2. Shaneck, M., Mahadevan, K., Kher, V., Kim, Y.: Remote software-based attestation for wireless sensors. In: Molva, R., Tsudik, G., Westhoff, D. (eds.) ESAS 2005. LNCS, vol. 3813, pp. 27–41. Springer, Heidelberg (2005)
3. Hill, J.L., Culler, D.E.: Mica:a wireless platform for deeply embedded networks (2002)
4. Coppersmith, D., Jakobsson, M.: Almost optimal hash sequence traversal. In: Financial Cryptography 02 (2002)
5. Atmel: Atmel 8-bit avr microcontroller with 16k bytes in-system programmable flash (2007)
6. Atmel: Atmel 8-bit avr microcontroller with 128k bytes in-system programmable flash. (2006)
7. Corporation, M.: Telos revision b. preliminary datasheet (2004)
8. Atmel: Atmel 4-megabit 2.5-volt or 2.7-volt dataflash (2005)
9. Ganesan, P., Venugopalan, R., Peddabachagari, P., Dean, A., Mueller, F., Sichitiu, M.: Analyzing and modeling encryption overhead for sensor network nodes. In: WSNA '03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications, San Diego, CA, USA, pp. 151–159. ACM Press, New York (2003)