Distributed Software-based Attestation for Node Compromise Detection in Sensor Networks

Yi Yang, Xinran Wang, Sencun Zhu, and Guohong Cao
Department of Computer Science & Engineering
The Pennsylvania State University
University Park, PA 16802
Email: {yy5, xinrwang, szhu, gcao}@cse.psu.edu

Abstract

Sensors that operate in an unattended, harsh or hostile environment are vulnerable to compromises because their low costs preclude the use of expensive tamper-resistant hardware. Thus, an adversary may reprogram them with malicious code to launch various insider attacks.

Based on verifying the genuineness of the running program, we propose two distributed software-based attestation schemes that are well tailored for sensor networks. These schemes are based on a pseudorandom noise generation mechanism and a lightweight block-based pseudorandom memory traversal algorithm. Each node is loaded with pseudorandom noise in its empty program memory before deployment, and later on multiple neighbors of a suspicious node collaborate to verify the integrity of the code running on this node in a distributed manner. Our analysis and simulation show that these schemes achieve high detection rate even when multiple compromised neighbors collude in an attestation process.

1. Introduction

Sensors that operate in an unattended, harsh or hostile environment often suffer from break-in compromises, because their low costs do not allow the use of expensive tamper-resistant hardware. Besides the exposure of secret information (e.g., cryptographic keys), compromised sensors may be reprogrammed with malicious code to launch all kinds of insider attacks. Very desirably, if we can identify and further revoke those corrupted nodes in a timely manner, the potential damages caused by them could be minimized. Intrusion detection mechanisms such as watchdog [9] and reputation-based system [5] have been proposed to detect abnormal nodes. However, these behavior-based detections are error-prone, because they rely on accurate observation and reasoning of node's misbehavior.

Recently, several software-based attestation techniques [18, 17, 15, 11] have been proposed to verify the integrity of the code running on an embedded device (e.g., sensor) without physical access to the device or assistance of secure hardware. Basically, these techniques use a challenge-response protocol between a trusted external verifier and an interrogated device. Take SWATT [15] as an example. A verifier in the transmission range of the device sends a randomly generated number as the challenge. Upon receiving this challenge, the interrogated device traverses its memory in a pseudorandom fashion while recursively computing a cryptographic checksum over each traversed memory space, and responds to the verifier with the final checksum. The verifier can validate the result because it knows the expected memory image, so that it can locally precompute the correct answer and estimate an upper bound for the response time. The pseudorandom memory traversal algorithm is designed in such a way that a tampered device either responds with a wrong answer or takes distinguishable longer time than a legitimate device does, leaving itself being detected.

Current software-based attestation techniques, however, cannot be directly applied in sensor networks due to several limitations. For example, in unattended sensor networks, it is not always possible to provide a trusted mobile verifier to enter the transmission range of a suspicious node due to safety and security concerns. As an alternative, since we trust the base station (BS) if there is one, the BS may naturally become a remote verifier. However, the response time evaluated by the BS could be affected by such factors as network channel collision. Moreover, it is unclear when and how to trigger this attestation process; that is, how could a verifier know when to attest which nodes? Even if there are reliable reporting mechanisms, the dispatch and authentication of a mobile verifier or the message propagation may cause some delay, during which an adversary probably has launched attacks or even subverted the whole network through those compromised nodes.

To address the above challenges, we propose two dis-

tributed software-based attestation schemes, based on the following observations. To reprogram a sensor without being caught in an attestation, the attacker needs to keep a copy of the original code somewhere in its memory space; also, for a compromised node, its neighbors are the first and direct observers of its suspicious behaviors. Accordingly, first noises (pseudorandom numbers) are filled into the empty program memory of each node before deployment. Then, either the seed for generating the noise (as of Scheme I) is distributed to multiple neighbors based on threshold secret sharing [16], or a random digest of the program memory content (as of Scheme II) is assigned to each neighbor. When an attestation is triggered, multiple neighbors of a suspicious node collaborate in a challenge-response process and make a decision regarding the trustworthiness of this node in a distributed manner.

Our distributed schemes differ from the previous ones in that (1) they only involve regular neighboring nodes and no trusted verifier or BS is included, so the attestation could be finished in a timely and distributed manner, and (2) they do not rely on response time difference to distinguish between a benign node and a compromised node. These nice features make them very attractive for unattended sensor networks where a BS or trusted verifier could be the single point of failure from both security and operation perspectives. For each scheme we perform extensive security analysis and performance evaluation, which shows that our schemes are very efficient and effective in detecting program memory changes of compromised sensors.

The remainder of this paper is organized as follows. Section 2 describes the related work. Then, Section 3 states our system model and assumptions. Block-based memory traversal algorithm is presented in Section 4. In Section 5, we propose two distributed software-based attestation schemes and analyze their security properties. Performance evaluation and comparison are given in Section 6. After that, some issues are discussed in Section 7. Finally, we summarize our work and discuss future work in Section 8.

2. Related Work

Assuming that program and associated data are stored in continuous memory locations, Spinellis [18] used cryptographic hashes computed over two overlapped random memory ranges (they together cover the entire memory space) to check the integrity of installed software in a machine. Shaneck et al. [17] proposed remote software attestation by sending a piece of attestation code to a sensor node. The attestation code is obfuscated to make static analysis of the code difficult for adversaries. Though interesting, it is however unclear how this idea could be implemented in real sensors. In PIV (Program Integrity Verification) [11], a new hash algorithm is generated for each attestation and verification servers are widely distributed to verify hashes received from attested nodes. Different from this work, our schemes only involve regular nodes.

Seshadri et al. [15] proposed SWATT in which a pseudorandom memory traversal algorithm is used to compute the checksum over all the traversed memory cells. According to Coupon Collector's Problem [10], as long as this process iterates for more than $O(m \ln m)$ times where m is the memory size, the entire memory space of interrogated node can be covered with a high probability. Our block-based memory traversal algorithm improves the cell-based algorithm of SWATT in that for each traversal a block of cells rather than one cell are accessed. Hence, less iterations and computational cost are involved in the attested node. More important, based on this algorithm we propose two distributed schemes where an attestation is fulfilled by multiple neighbors instead of a trustable verifier as in SWATT.

Software-based attestation could also be used in computer systems [14]. Seshadri et al. [14] took advantage of SWATT-like pseudorandom memory traversal to construct a trusted computing base, based on which hashes of executables were measured to provide untampered code execution. Moreover, Wurster et al. [19] found that in an architecture where code and data reads are separated, checksumming-based tamper resistance is subject to such an attack that checksum is computed over the original program whereas the code that gets executed is actually the malicious version. Similar situation exists in sensor networks where an adversary keeps a copy of the original program in sensor's empty memory which is accessible during checksum computations.

3. System Model and Assumptions

Network Model We consider a sensor network composed of a large quantity of low-cost sensors that are constrained by scarce resources in power, computation, communication and storage (e.g., 128KB program memory and 4 KB RAM in the case of Mica2 motes). We assume that sensors are densely deployed in the network so that each node has multiple immediate neighbors and the neighbors can communicate with each other. There is a way for sensors to discover neighbors (e.g., by sending a HELLO message [22]) and to establish a pairwise key shared with every neighbor [8].

Attack Model We assume that an attacker can capture a (small) fraction of sensors, reprogram them with malicious code, and redeploy them back into the network. Since in our setting all the nodes are equally important and vulnerable to compromises, we assume that every node is equally likely to be compromised. Like all the previous schemes [18, 17, 15, 11], we assume that the attacker does not enlarge the sensor's memory; unlike the time delay based attestation schemes [17, 15] that further assume the processor speed and memory access rate are not increased, our schemes do not make such assumptions.

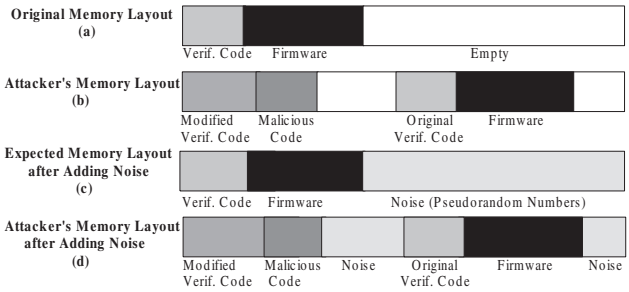


Figure 1. Program memory layout before and after adding noise.

Once compromised, sensors are under the full control of the attacker. They may launch various attacks, including passive attacks such as eavesdropping, and active attacks such as false data injection. Detection of node's abnormal behavior (e.g., by watchdog [9]) may trigger our attestation protocol; however, it is not necessarily the only condition. For example, our attestation protocol may be executed with a reasonably large (e.g., days) and random trigger interval to reduce the attestation cost and also to prevent the time-of-check-to-time-of-use (TOCTTOU) attack [1]. Furthermore, we assume the compromised nodes in the same neighborhood may collude by sharing their knowledge and resources during an attestation.

Note that our main concern is sensor's program space instead of data space, because of the following reasons. Data space is normally used to store current execution information, such as program stack and temporary data. Therefore, certain parts of the data memory cannot be overwritten by the attacker. Otherwise, it may cause the sensor program to crash. This not only largely reduces the amount of available space for the attacker, but also requires the attacker to be very careful when considering where to put the copy of the original code. If we also take into account the fact that the size of data space is typically two orders of magnitude smaller than that of the program space, we assume that the attacker will not put original code to sensor's data space.

4. Preliminaries

Besides the normal functioning code, a piece of verification code is also loaded at the beginning of a sensor's program memory to support code attestation. As shown in Figure 1(a)[1], the original layout of memory contains the verification code, firmware and some empty space. The empty space exists because sensor nodes have fixed-size code space that is usually not filled completely. Figure 1(b) illustrates the modified layout of sensor's memory by an attacker, where some malicious code takes over the memory space of the original code and the latter is moved to the empty space. Our idea to prevent this attack is to proactively fill the empty space of every node with pseudorandom numbers (also called *noise*, Figure 1(c)) before node deployment. Noises are derived from a secret seed that is unique for each node. The attacker does not know the seed for generating the noise, thus is unable to compress the noise to make empty space for his own malicious code. If the attacker moves the original code to overwrite the noise space (Figure 1(d)), our attestation schemes will guarantee that once challenged the probability for the compromised node to return a correct checksum is negligible.

In the following, we first present a specific noise generation method, then describe a block-based pseudorandom memory traversal algorithm as the underlying memory traversal mechanism for our distributed software-based attestation schemes.

Noise Generation To generate noises for our purpose, we use the block cipher RC5 running in *CTR (counter) mode* as Pseudo Random Number Generator (PRNG). The design principle is for the attestor's convenience in reconstructing the attested node's memory image. This is done by taking a seed as the input and each time encrypting an incremental counter, i.e., first encrypting a 0, then encrypting a 1, and so on. We call this seed to the PRNG as *noise-generation seed*, which contains information about all the generated noise. Since each time we are outputted with an 8-byte noise, we only need $m'/8$ counters to generate all the noise, if m' is empty memory size (in byte) to be filled.

Counter mode offers a nice property that the (1-byte) noise in each memory cell is individually accessible by directly encrypting the right counter and finding out the correct offset in the corresponding 8-byte value. As such, in the block-based traversal algorithm introduced below, the attestor keep only three variables in its data space during memory traversal: the current checksum, the block value, and the accessed memory cell within the block. This is highly desirable because an attestor consumes little data space to locally generate a correct answer for comparison. Note if noise is generated by *CBC mode*, i.e., first encrypting the seed, then repeatedly applying encryption over the previous output, the attestor will spend a lot of memory in constructing the expected memory image of attested node because all the noise must be generated sequentially. This is infeasible in our schemes since an attestor is just a regular sensor whose own program space is filled up with noise and data space is limited.

Block-based Pseudorandom Memory Traversal There are two types of memory traversals, sequential and pseudorandom. In a sequential traversal scheme [11], a checksum is computed over a block of memory and a checksum vector over all the blocks reflects integrity of the entire memory. This scheme, however, is ineffective against collusion among nodes. Here is an example (Figure 2). Let the original memory of a node u be divided into two halves, M_1 and M_2. Node u may store M_1 and resort to its neighbor v

[1]This is a schematic figure. The actual layout may be slightly different.

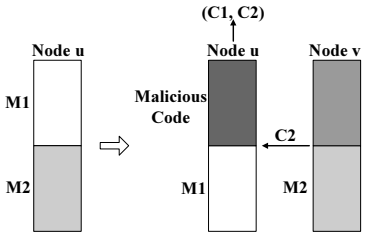


Figure 2. Collusion between node u and v in sequential memory traversal.

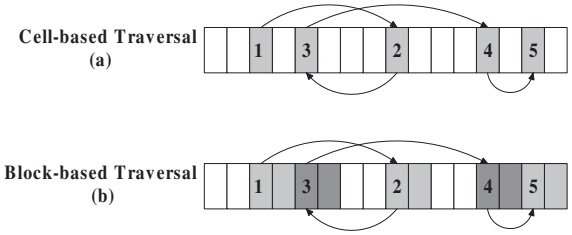


Figure 3. Cell-based and block-based ($b = 2$) pseudorandom memory traversal.

to store the other half M_2. Thus, node u could use the other half for malicious code. When challenged, node v computes checksum C_2 over M_2, then passes C_2 to node u, which locally computes C_1 over M_1 and returns the correct result (C_1, C_2).

Pseudorandom memory traversal [15] is effective to defend against the above attack because the order of memory access is unpredictable to the attacker. However, it has much higher traversal overhead than a sequential algorithm, because each memory cell is accessed multiple times to cover the entire memory space of an attested node with a high probability. Based on the Coupon Collector's problem, on average $O(m \ln m)$ traversals are needed for the memory size of m; that is, $O(\ln m)$ traversals per cell. For Mica2 mote [3] with $128KB$ program space, this means 11.7 traversals per cell. We call this algorithm *cell-based* pseudorandom memory traversal, as shown in Figure 3(a). To reduce the overhead, we propose a *block-based* pseudorandom memory traversal algorithm (Algorithm 1), in which we traverse memory block-by-block instead of cell-by-cell and efficient 'XOR' operations are executed within blocks. As shown in Figure 3(b), each block has b continuous memory cells.

Algorithm 1 Block-based Memory Traversal Algorithm

Input: i_t-number of iterations, j-current byte of checksum, b-block size, m-memory size, A-traversed memory address;
Output: a checksum $C(C_0, \cdots, C_7)$;
Procedure:
1: C is initialized as a 64-bit random number;
2: j is initialized to point to C_0, the first byte of C;
3: **for** $i = 1$ to $i_t/4$ **do**
4: $(A_0, A_1, A_2, A_3) = RC5_i$;
5: **for** $k = 0$ to 3 **do**
6: $C_j = C_j + (Mem[A_k \pmod m] \oplus \cdots \oplus Mem[A_k + b - 1 \pmod m])$;
7: $j = (j+1) \mod 8$;
8: **end for**
9: **end for**
10: return C;

In Algorithm 1, the PRNG used for memory traversal is RC5 in counter mode. Taking a seed, each time RC5 outputs an 8-byte value which can be treated as four memory addresses, since each memory address is 2-byte (e.g., Mica2 mote). Therefore, from one RC5 computation we get 4 addresses for memory traversals, so that the total number of RC5 computations is actually $i_t/4$, if i_t is the number of traversal iterations to cover the whole memory space. We call this seed to the PRNG as *memory-traversal seed*, which determines the order of pseudorandom memory traversals. Using the same memory-traversal seed, in the following attestation, the attestor and the attested node should get the same checksum results over the same memory content. Every time we construct a 16-bit pseudorandom address for memory read, we update the corresponding 8-bit checksum based on the bitwise 'XOR' result of a block of cells. This procedure is repeated until sufficient number of iterations are finished to access each memory cell at least once.

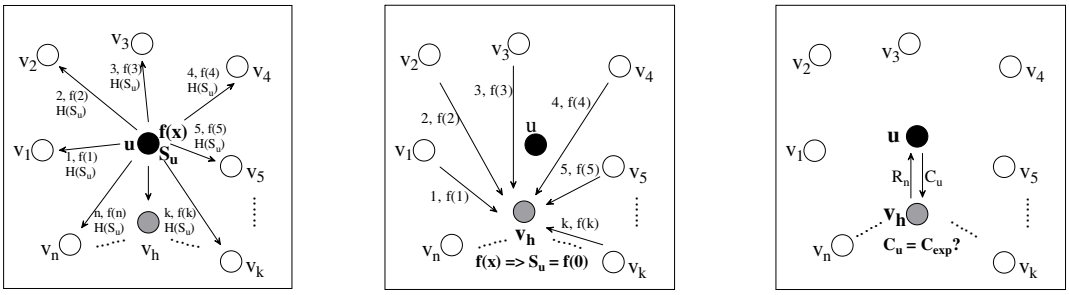
We have a theorem governing the number of iterations needed in Algorithm 1. (All the proofs are available in [20].)

Theorem 4.1 *In block-based pseudorandom memory traversal, suppose b is block size, m is memory size, and random variable Y represents the number of traversal iterations needed to cover each memory cell at least once, then* $E(Y) = O(\frac{m \ln m}{b})$ *and* $Pr[Y > \frac{cm \ln m}{b}] \leq m^{1-c}$, *where c is a constant factor.*

From Theorem 4.1, we can see that $i_t \approx O(\frac{m \ln m}{b})$. Also, this theorem indicates that if we traverse the node's memory in a block of b cells at a time, then the number of traversal iterations is $1/b$ of that in cell-based traversal to ensure each cell has the same level of possibility being accessed. Although the total number of cells to be traversed is still the same, the computational overhead is reduced through efficient 'XOR' computations. Note that cell-based traversal is a special case of block-based traversal where $b = 1$. This does not mean it is absolutely better if b is larger. If $b = m$ then we traverse all the memory cells sequentially by 'XOR' operations and update the checksum only once. This is less secure because there is a high probability of collision and an attacker may simply remember the XOR of all cells. Therefore, there is a tradeoff between performance and security, and we need choose an appropriate value of b. For Mica2 mote, every read of program memory returns 16 bytes instead of 1 byte, it is therefore recommended that we set b to be 16 (or its multiples).

5. Proposed Schemes

We propose two distributed software-based attestation schemes based on the block-based pseudorandom memory



(a) Share distribution (b) Seed recovery (c) Attestation

Figure 4. Scheme I: A Basic Threshold Secret Sharing Scheme.

traversal algorithm and analyze their security properties.

5.1. Scheme I: A Basic Threshold Secret Sharing Scheme

There are three basic steps in the first scheme: *noise generation*, *secret share distribution*, *secret seed recovery and attestation*. First, the empty memory space of each node, say u, is filled with pseudorandom numbers derived from a unique noise-generation seed S_u. This is finished offline before node deployment. Then, after node deployment, node u distributes one share of its noise-generation seed to each of its neighbors. Later, when an attestation is triggered against node u, neighbors collaborate to recover S_u, reconstruct the memory image of node u, and attest its memory content through pseudorandom memory traversals. Since we already introduce the technique for noise generation in the previous section, next we discuss details of secret share distribution as well as secret seed recovery and attestation.

Secret Share Distribution We assume that it takes an attacker at least a time period T_{min} to compromise a sensor [22, 4]. Besides the noise that are filled into the empty memory space, the noise-generation seed is also preloaded into sensor's memory. After a sensor node u is deployed, e.g., by aerial scattering, it does the following:

- It discovers neighbors (e.g., by sending a HELLO message) and meanwhile it starts a timer which will expire after T_{min}. It also establishes a pairwise key with each neighbor on the fly[2].
- It splits S_u into multiple shares and sends a separate share to each neighbor; a hash value $H(S_u)$ computed by one-way hash function H is also included in the message, which enables every neighbor to easily verify the correctness of a recovered seed in the future while preventing a neighbor from deriving S_u from $H(S_u)$.
- When the timer expires, it removes S_u from memory.

If we rely on a trusted verifier or BS, it is relatively easy to recover the memory image and validate the response. In our setting, however, no nodes are absolutely trustable. To address this issue, we adopt Shamir's (k,n) threshold secret sharing [16, 2] for every node to distribute shares of the noise-generation seed to multiple neighbors, which will later collaborate in recovering the seed for attestation. In our case, n is the number of neighbors and k ($1 \leq k \leq n$) is a system threshold which relates to the network density and reflects a tradeoff between security and performance. More specifically, node u distributes secret shares to its neighbors as follows. First, node u randomly picks up $k-1$ constants denoted by $a_1, a_2, \ldots, a_{k-1}$ in a prime finite field Z_p and constructs a univariate polynomial $f(x) = S_u + a_1 x + a_2 x^2 + \ldots + a_{k-1} x^{k-1}$, where $S_u = f(0)$ is the noise-generation seed. Then, as shown in Figure 4(a), a tuple (i, f(i)) is distributed to neighbor v_i ($1 \leq i \leq n$) securely.

Note that we assume there exists a lower bound on the time interval T_{min} that is necessary for an adversary to compromise a sensor node [22], and that the time T_{est} for a newly deployed sensor node to discover its immediate neighbors and to distribute secret shares of noise-generation seed is smaller than T_{min}. Based on real experiments, Deng et al. [4] showed that it is possible for an experienced attacker to obtain copies of all the memory and data of a Mica2 mote in tens of seconds or minutes after a node is captured. Zhu et al. [21] showed through experiments that T_{est} is in the order of several seconds for a network of a reasonable node density (up to 20 neighbors). Therefore, we believe that it is a reasonable assumption in practice that $T_{min} > T_{est}$. As a result, within time T_{min} after deployment, each node stores a pairwise key, a secret share and a hash value for every neighbor. These are kept in data space, so we do not need to verify this part of node u's memory.

Secret Seed Recovery and Attestation An attestation is triggered if a sufficient number (e.g., more than half) [3] of neighbors detect the abnormal behavior of node u. Then, all the neighbors $v_1, v_2, \ldots, v_n$ of node u elect a cluster head[4], denoted by $v_h (1 \leq h \leq n)$, among themselves based on an

[2]Since then, all the messages transmitted between two nodes will be encrypted by corresponding pairwise keys, unless mentioned otherwise.

[3]In this way, few number of colluded malicious neighbors cannot accuse an honest node u by exhausting its energy for attestation.

[4]It is called cluster head hereafter although there are no topological clusters here.

appropriate election algorithm [13]. The role of cluster head is rotated among all the neighbors for each attestation due to security and performance (load balance) reasons. The current cluster head v_h sends an authenticated challenge R_n, which is a random number, to node u. While node u computes the response over its own memory space by i_t traversals based on our block-based algorithm with R_n as the memory-traversal seed, v_h does the following:

- It collects k secret shares from neighbors of node u, so that the polynomial $f(x)$ is uniquely determined (by Lagrange interpolation) and S_u is recovered by evaluating this polynomial at 0, as shown in Figure 4(b).
- It verifies whether the recovered noise-generation seed is valid by comparing its hash value with the locally kept $H(S_u)$. If they do not match, it requests secret shares from another set of k neighbors until a correct noise-generation seed is reconstructed.
- It locally computes the expected memory traversal checksum C_{exp}, based on S_u and R_n. Specifically, from the recovered noise-generation seed S_u, it knows the expected memory image of node u (our counter-mode based noise generation enables it to readily regenerate the noise for each empty space traversal upon node u). Then, taking R_n as the memory-traversal seed, our block-based memory traversal algorithm outputs the correct checksum C_{exp} after i_t traversals.
- It compares C_{exp} with the responded checksum C_u from node u and concludes that the interrogated node u has been compromised if these two checksums are different, as shown in Figure 4(c).

Security Analysis We analyze the security properties of Scheme I in terms of the detection rate (i.e., the probability for neighbors to successfully detect a compromised node). First, we have a lemma discussing under what conditions Scheme I is able to detect a compromised node.

Lemma 5.1 *Scheme I is able to successfully detect a compromised node u, if: (1) cluster head is trustable; (2) cluster head obtains $\geq k$ correct shares of noise-generation seed S_u from neighbors of node u; (3) the attacker does not obtain $\geq k$ shares to recover the seed S_u.*

Next, we derive the detection rate of Scheme I, based on Lemma 5.1.

Theorem 5.2 *Assuming that the probability for each node in the network to be compromised is the same and equals to $p_0 (0 < p_0 < 1)$, then the detection rate of Scheme I is*

$$P_{bs}(k,n) = \begin{cases} \sum_{i=k-1}^{n-1} \binom{n-1}{i} (1-p_0)^{i+1} p_0^{n-1-i}, & \text{if } n < 2k \\ \sum_{i=n-k}^{n-1} \binom{n-1}{i} (1-p_0)^{i+1} p_0^{n-1-i}, & \text{if } n \geq 2k. \end{cases}$$

When $n = 15$, the detection rate of Scheme I is shown in Figure 5, from which we can see that if p_0 is larger (i.e., nodes in the network have a higher possibility to be compromised), then P_{bs} is lower. In addition, suppose p_0 is fixed,

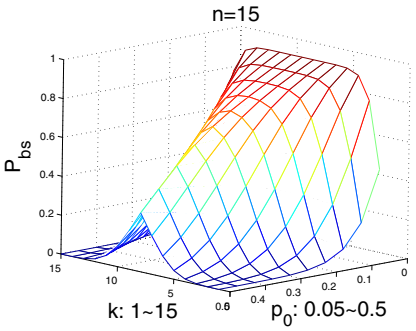


Figure 5. Detection rate of Scheme I.

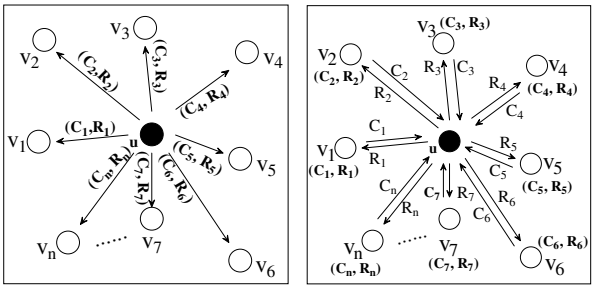
then P_{bs} has a peak value with $k = \frac{n+1}{2}$ (if n is an odd integer) or $k = \lceil \frac{n+1}{2} \rceil, \lfloor \frac{n+1}{2} \rfloor$ (if n is an even integer), and P_{bs} is symmetric when k is larger or smaller than these values. This means when we choose the values of k and n we should make k as close to half of n as possible so that Scheme I has a higher detection rate. For example, suppose p_0 is 0.05 and $n = 15$, the detection rates when k equals to 7, 8 are both about 95%.

Scheme I could be enhanced if we are able to prevent the attacker from obtaining more than k secret shares. We can adopt proactive secret sharing [6] to periodically update shares of S_u without changing S_u itself. As a result, even if an attacker has managed to obtain as many as $k-1$ secret shares (the threshold is k), a proactive share renewal process will render them useless. Specifically, after a certain time period, a $(k-1)$-degree polynomial $g(x)$ is randomly selected by node u over Z_p, satisfying that $g(0) = 0$. After the neighbor v_i receives the distributed share $(i, g(i))$, it renews the share it kept by $(i, f(i) + g(i))$. Since $f(0) + g(0) = S_u + 0 = S_u$, the noise-generation seed S_u is not changed upon this renewal.

We note that in this scheme a compromised neighbor may contribute an erroneous share. The cluster head will need to pick another set of k shares to redo the seed recovery, thus consuming additional energy. This problem could be solved if a node has some additional data memory space. For example, besides storing $H(S_u)$, each neighbor of node u also stores a hash of every secret share. This will allow it to easily verify the received shares from other neighbors when it becomes the cluster head, thus compromised neighbors are deterred from contributing wrong shares. We will investigate more practical tradeoffs in our future work.

5.2. Scheme II: A Majority Voting Based Attestation Scheme

In Scheme I, the cluster head can reconstruct the noise-generation seed and hence has a way to know the expected memory content of the attested node. Based on this knowledge, the cluster head is able to send a random challenge at will for each attestation. This prevents the attested node from precomputing the response based on prediction. How-



(a) Information distribution (b) Attestation

Figure 6. Scheme II: A Majority Voting based Attestation Scheme.

ever, as we have seen, the compromise of the cluster head may result in a wrong decision about the attested node. Also, once a noise-generation seed is disclosed (e.g., due to neighbor compromises), an attacker may replace the noise with malicious code in a sensor that is later compromised.

To address the above problems, we consider two strategies. First, instead of relying on some specific cluster head to make decisions, we make use of a majority voting scheme among neighbors. Second, instead of distributing and recovering the noise-generation seed, each neighbor is distributed with and keeps a challenge as well as the corresponding response. During an attestation, each neighbor sends the challenge and waits for the response from the attested node. If the received response is different from the local one, then the attested node is considered compromised.

The advantages of this scheme are three-fold. First, neighbors do not need compute any responses locally, greatly reducing the computational overhead involved in the attestors. Second, although a compromised neighbor knows which cells it will traverse, it does not know which cells will be traversed by other neighbors. Hence, the attacker cannot decide which cells are safe to modify for sure. Third, based on majority voting, an innocent node will not be identified as compromised due to one or a few compromised neighbors. Next, we provide the details of information distribution and attestation in this scheme.

Information Distribution Before deployment, noise is preloaded into each node's empty memory space. Each node is also preloaded with n tuples of (C_i, R_i) $(1 \leq i \leq n$, n is no less than the estimated maximum number of neighbors), where C_i is a challenge and R_i is the corresponding response. Each tuple is generated by an offline server as follows. Taking a random challenge C_i as the memory-traversal seed, node u's memory is traversed i_t times based on our block-based pseudorandom memory traversal algorithm and a checksum R_i is returned. After deployment, every node (say u) discovers neighbors, securely delivers to each neighbor $v_i(1 \leq i \leq n)$ a randomly picked tuple (shown in Figure 6(a)) within T_{min}, and finally erases all the tuples after T_{min}.

Note that the number of traversal iterations i_t in this scheme is different from that in the previous scheme. Suppose that the memory size of node u is m, then i_t should be $O(\frac{m \ln m}{bn})$ or above, so that all the neighbors may corporately traverse each of node u's memory cells at least once.

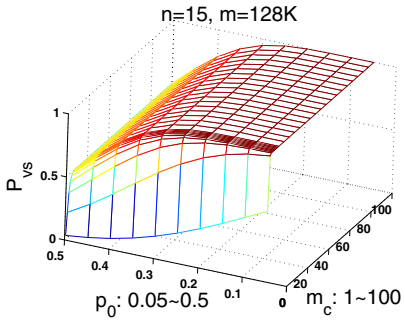
Attestation Later on, if more than half of neighbors agree to attest node u, neighbors attest node u in sequence (to prevent channel collisions), as shown in Figure 6(b). Specifically, each neighbor v_i securely sends node u its challenge C_i and waits for the response. Then, taking the challenge C_i, node u traverses its memory accordingly based on the block-based pseudorandom memory traversal algorithm with i_t iterations, and reports the resulted checksum as the response. Note that sequence numbers must be added to both the challenge and response messages before encryptions to prevent replay attacks. After that, neighbor v_i compares this checksum with the one it keeps locally (R_i) and makes its own decision about node u. Finally, if the number of neighbors who have negative opinions exceeds the majority (i.e., $\lceil \frac{n+1}{2} \rceil$), then node u will be identified as compromised. In this way, each neighbor of node u has its independent judgement about node u's memory integrity.

Security Analysis According to the Byzantine generals problem [12, 7], if a compromised neighbor cannot modify opinions from other neighbors (i.e., a faulty neighbor may lie on its own behalf, or refuse to relay results received from others, but it cannot alter other's results without betraying itself as faulty), then all the honest neighbors can finally reach an agreement about node u's code memory integrity by employing authentication mechanisms. In our case, the pairwise keys shared and known only between two nodes could provide the same functionality.

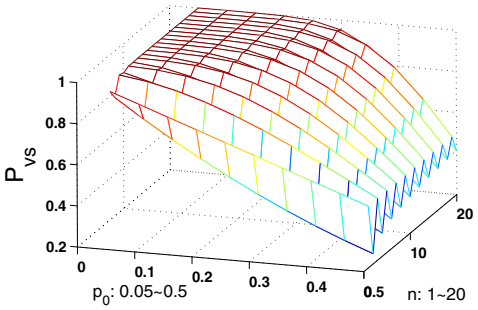
The choice of i_t (i.e., the number of memory traversal iterations) reflects a tradeoff between performance and accuracy: if i_t is larger, then the overhead is higher, but the decisions from neighbors is more reliable, because if $i_t = O(\frac{m \ln m}{b})$, then every neighbor will make node u traverse each memory cell at least once with a high probability and hence obtain a sound opinion about node u's memory. However, the total traversal cost involved in node u will be n times of that in Scheme I. On the other hand, if we reduce the number of iterations (e.g., to less than $O(\frac{m \ln m}{bn})$), it is possible that neighbors fail to detect the modified part of the attested node's code memory.

The detection rate of Scheme II is formalized by the following theorem.

Theorem 5.3 *Assuming that the probability for each node in the network to be compromised is the same and equals to $p_0(0 < p_0 < 1)$. For Scheme II with regard to node u, suppose m_c is the number of changed memory cells of node u, m is node u's memory size, and we choose $i_t = \frac{m \ln m}{bn}$, then the detection rate of Scheme II is $P_{vs}(n, m_c, m) = \sum_{i=\lceil \frac{n+1}{2} \rceil}^{n} [\binom{n}{i}(1-p_0)^i p_0^{n-i} \sum_{j=\lceil \frac{n+1}{2} \rceil}^{i} \binom{i}{j} p_h^j (1-p_h)^{i-j}]$, where $p_h = 1 - (\frac{m-m_c}{m})^{\frac{m \ln m}{n}}$ is the probability for an honest neigh-*



(a) P_{vs}: different m_c, fixed n



(b) P_{vs}: different n, fixed $m_c (> 0)$

Figure 7. Detection rate of Scheme II.

bor to detect the compromised node u.

The detection rate of Scheme II with different m_c and fixed n is shown in Figure 7(a). Even if the attacker changes only 3 bytes, P_{vs} is already increased to 99% when $p_0 = 0.05$. In this case, p_h is also as high as 90%. The detection rate of Scheme II with different n and fixed m_c can be seen in Figure 7(b). P_{vs} vibrates as the total number of neighbors n changes, since n alternates between odd and even numbers in majority voting.

6. Performance Evaluation

We first analyze the performance of our schemes, then we conduct simulations to show that our schemes can effectively detect changed memory content of sensor nodes.

6.1. Performance Analysis

We quantify the overhead of our schemes from three aspects: computation, communication and storage.

Computational Cost In Scheme I, for the attested node, there are n $(k-1)$-degree polynomial evaluations and one hash computation to distribute shares to n neighbors. Also, there are $O(\frac{m\ln m}{b})$ memory traversals involved during attestation. For neighbors, to recover and validate the noise-generation seed, there are one $(k-1)$-degree polynomial interpolation, one $(k-1)$-degree polynomial evaluation and one hash computation. In attestation, $O(\frac{m\ln m}{b})$ memory traversals are needed to cover the whole memory of the attested node. In Scheme II, since (challenge, response) pairs are generated offline, there are only $O(\frac{m\ln m}{b})$ memory traversals involved in the attested node.

Communication Overhead Suppose L_s, L_h and L_c are the lengths of a secret seed or share (they have the same length in our schemes), hash value, and checksum, respectively. In Scheme I, the message overhead includes: secret share distribution of $n(L_s+L_h)$, secret seed recovery by the cluster head of at least $(k-1)L_s$, and the attestation overhead of L_s+L_c. For Scheme II, the message overhead includes: challenge and response distribution of $n(L_s+L_c)$ and the attestation overhead of $n(L_s+L_c)$.

Storage Requirement In Scheme I, each node needs to store secret shares for its n neighbors with the storage cost of $n(L_s+L_h)$ bytes. Also, it costs neighbors of the attested node $n(L_s+L_h)$ bytes to store all the secret shares. In Scheme II, it costs neighbors $n(L_s+L_c)$ bytes to store all the challenges and responses for the attested node.

Comparison of Two Schemes Obviously, Scheme II has lower computation overhead than Scheme I. The length of checksum is 8-byte in our algorithm ($L_c = 8$ bytes). Suppose we choose L_s and L_h to be 8-byte, too. Then, the storage overhead of these two schemes are almost the same. Moreover, Scheme II has higher communication overhead than Scheme I.

From the above analysis and comparison, we have the following observation and conclusion. Scheme II has better security properties, but it also has higher communication overhead. Scheme I has lower communication overhead but relatively higher computational cost. As such, which one to use should be based on specific resource configurations and application requirements.

6.2. Simulation results

We conduct simulations to verify the effectiveness of our schemes in detecting memory cells changed by the attacker. The attacker in our simulation randomly selects a memory cell, beginning from which the attacker changes the content of m_c continuous memory cells. From the simulation results we find that in practice the overhead to detect changed memory cells can further be largely reduced because the number of traversal iterations needed to detect the modification is actually much less than expected. The number of iterations to access each memory cell at least once could be used to detect even one byte of memory change. In practice, however, the attacker needs many more continuous memory cells (e.g., several hundred bytes) to inject malicious code that can really do harm.

In the simulation of Scheme I, the size of modified memory content m_c is changed from 30 to 200 in bytes and each point obtained in the figure is the average value of 100 rounds. As shown in Figure 8, although the number of iterations needed to cover the entire memory space is $O(m\ln m) = 1505K$ when memory size m is $128K$, in Scheme I the actual traversal iterations for the cluster head

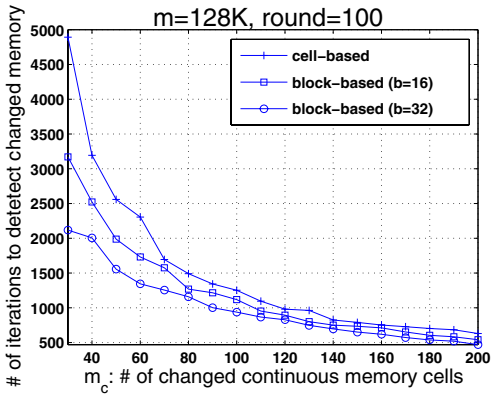


Figure 8. Number of iterations for cluster head to detect changed cells in Scheme I.

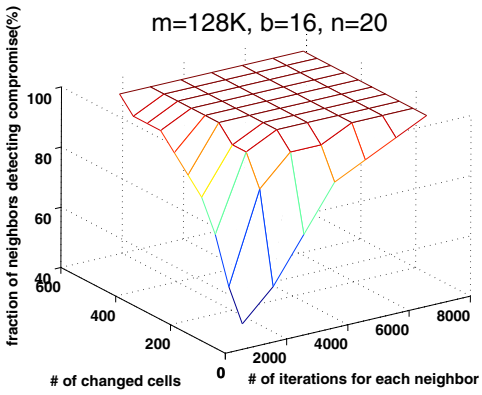


Figure 9. Fraction of neighbors successfully detecting changed cells in Scheme II.

to detect changed memory content with size as small as 30-byte is only about 3200 if block size $b = 16$. If we increase the block size to 32, then the number of iterations to detect this change is even smaller: about 2100 iterations to detect the changed 30 bytes, while in this case we need 4900 iterations in cell-based traversal.

In the simulation of Scheme II, we fix the memory size m to be $128K$, block size b to be 16, and the number of neighbors n to be 20. With the number of changed continuous memory cells varying from 50 to 500, we examine the fraction of neighbors that can successfully detect the modified memory part, under different number of traversal iterations for each neighbor (as shown in Figure 9). When the size of changed memory is small (e.g. $m_c = 50$), the value of $\frac{O(m \ln m)}{bn} = 6272$ gives a good lower bound to make sure every neighbor can detect the change. Nevertheless, if we have some knowledge on the number of changed memory cells, we may adjust or even reduce the number of memory traversal iterations accordingly. In fact, when 500 bytes of memory have been modified, there are only 1000 traversal iterations needed to guarantee that all the neighbors can detect this modification. In this case, to tolerate a certain number of compromised neighbors (e.g., less than half), majority voting is a good idea as long as more than half of neighbors are honest.

7. Further Discussions

Next, we discuss some important issues regarding our schemes.

Inaccuracy in Neighbor Number Estimation In Scheme II, our discussion implicitly assumed that a node knows in advance the number of neighbors n when preloaded the challenge/response pairs. In real networks, the actual number of neighbors may be different from what we have predicted. As such, we should adjust i_t dynamically so that the cell traversal probability and the number of iterations do not vary much. One simple approach to addressing this is as follows. Suppose we know that the maximum number of neighbors in the network is $n_{max} = 16$, then node u is preloaded with n_{max} challenge/response tuples. If the actual number $n = 9$, node u may send $\lceil \frac{n_{max}}{n} \rceil = 2$ tuples to each neighbor. Later on during an attestation, u will make two traversals for each neighbor. Note that after node u erases these n_{max} tuples, some empty space will be created. As such, instead of zeroizing the space, node u could fill it with some noises that are generated on-the-fly based on the noise generation seed and then erase the seed. For correctness, the offline generation of n_{max} responses should be based on these noises, not the challenge/response tuples.

Topology Changes In the description of our schemes, we mainly consider a static network model. For some sensor networks, nodes may be added and die during the network lifetime. Here we discuss the influence of network topology changes on our schemes.

In general, node removal is less a concern in our schemes than node addition. Scheme I works well if some neighbors die, as long as the number of neighbors is still more than the threshold value. The removal of neighbors is not a problem in Scheme II as long as the fraction of honest neighbors is still larger than one half. When a new node is added, it discovers its neighbors and then distributes its shares to the neighbors, following the same procedure as before. The challenge arises from getting shares from its neighbors which have done their secret share distributions. According to our protocols, these neighbors have erased their own noise seeds, thus they do not have any additional share to give out to this new node.

To (partially) address the problem, we may apply the following idea. Node deployment is divided into multiple intervals and there is an interval key for each interval. Nodes deployed in interval T_i carry the interval key K_i as well as all the past interval keys. When a node is deployed, it generates more shares than its actual number of neighbors; the extra shares could be encrypted with its interval key and

store locally. The node erases the interval keys after T_{min}, thus it cannot decrypt the share itself. However, a new node deployed either in the same time interval or later carries the interval key used in the encryption, so it can decrypt the share. In this way, a new node may collect one encrypted share from each neighbor and joins the neighbor for later attestation operations. Note that the security assumption here is the same as in our schemes; that is, a node will not be compromised within T_{min}. We will study this approach in more details and investigate more secure solutions in our future work.

8. Conclusion and Future work

The detection of node compromise is a critical but challenging problem for resource-constrained sensors deployed in an unattended or hostile environment. Recent work on software-based code attestation has shed light on accurately identifying compromised nodes. However, they are not readily applied into regular sensor networks due to one or another limitations. We have presented two distributed schemes towards making software-based attestation more practical. Our schemes do not depend on response time measurement by mobile verifiers or the base station. Instead, neighbors of a suspicious node collaborate in the attestation process to make a joint decision.

To the best of our knowledge, this is the first paper to address the compromised node detection issue in a totally distributed way. As an initial work, we do not expect to solve all the problems. In the future, we will further investigate solutions for noise-generation seed update and network topology change (i.e., nodes join and leave the network). Another issue we have not addressed yet is how to schedule the message transmissions of neighbors (for secret share distribution and collection) to minimize channel collision. Finally, we will study how to apply our schemes to different sensor memory architectures.

Acknowledgment We thank the anonymous reviewers for their valuable comments. This work was supported by US Army Research Office (ARO) under grant W911NF-05-1-0270, and by the National Science Foundation (NSF) under grants CNS-0524156, CNS-0627382, and CAREER NSF-0643906.

References

[1] Time-of-check-to-time-of-use(TOCTTOU). http://en.wikipedia.org/wiki/Time-of-check-to-time-of-use.

[2] P. C. v. O. Alfred J. Menezes and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. http://www.cacr.math.uwaterloo.ca/hac/.

[3] Crossbow Technology, Inc. *Mica Motes*. http://www.xbow.com.

[4] J. Deng, R. Han, and S. Mishra. A practical study of transitory master key establishment for wireless sensor networks. In *SecureComm 2005*, pages 289–299, September 2005.

[5] S. Ganeriwal and M. Srivastava. Reputation-based framework for high integrity sensor networks. In *SASN'04*, 2004.

[6] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *CRYPTO '95*, pages 339–352, 1995.

[7] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. In *ACM Transactions on Programming Languages and Systems*, volume 4, pages 382–401, July 1982.

[8] D. Liu, P. Ning, and R. Li. Establishing pairwise keys in distributed sensor networks. *ACM Transactions on Information and System Security*, 8(1):41–77, February 2005.

[9] S. Marti, T. J. Giuli, K. Lai, and M. Baker. Mitigating routing misbehavior in mobile ad hoc networks. In *MobiCom*, pages 255–265, 2000.

[10] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[11] T. Park and K. G. Shin. Soft tamper-proofing via program integrity verification in wireless sensor networks. *IEEE Trans. Mob. Comput.*, 4(3):297–309, 2005.

[12] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. In *Journal of ACM*, volume 27, pages 228–234, April 1980.

[13] M. Pirretti, S. Zhu, V. Narayanan, P. McDaniel, M. Kandemir, and R. Brooks. The sleep deprivation attack in sensor networks: analysis and methods of defense. In *ICA DSN*, October 2005.

[14] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *SOSP*, pages 1–15, October 2005.

[15] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: SoftWare-based ATTestation for embedded devices. In *IEEE Symposium on Security and Privacy*, 2004.

[16] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[17] M. Shaneck, K. Mahadevan, V. Kher, and Y. Kim. Remote software-based attestation for wireless sensors. In *ESAS*, July 2005.

[18] D. Spinellis. Reflection as a mechanism for software integrity verification. *ACM Trans. Inf. Syst. Secur.*, 3(1), 2000.

[19] G. Wurster, P. C. van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *SP' 2005*, pages 127–138.

[20] Y. Yang, X. Wang, S. Zhu, and G. Cao. Distributed software-based attestation for node compromise detection in sensor networks. Technical report, Department of Computer Science and Engineering, The Pennsylvania State University, 2007. http://www.cse.psu.edu/ yy5.

[21] S. Zhu, S. Setia, and S. Jajodia. LEAP+: Efficient security mechanisms for large-scale distributed sensor networks. http://www.cse.psu.edu/ szhu/papers/leap.pdf, to appear in ACM TOSN.

[22] S. Zhu, S. Setia, and S. Jajodia. LEAP: efficient security mechanisms for large-scale distributed sensor networks. In *CCS '03*, pages 62–72, 2003.