Secure random number generation in wireless sensor networks

G. Lo Re, F. Milazzo, M. Ortolani

In Journal of Concurrency and Computation: Practice and Experience

NDS Research Group - Networking and Distributed Systems
University of Palermo, ITALY
http://www.dicgim.unipa.it/networks

Secure Random Number Generation in Wireless Sensor Networks

Giuseppe Lo Re giuseppe.lore@unipa.it
Fabrizio Milazzo fabrizio.milazzo@unipa.it
Marco Ortolani marco.ortolani@unipa.it

Abstract

The increasing adoption of Wireless Sensor Networks as a flexible and inexpensive tool for the most diverse applications, ranging from environmental monitoring to home automation, has raised more and more attention to the issues related to the design of specifically customized security mechanisms. The scarcity of computational, storage, and bandwidth resources cannot definitely be disregarded in such context, and this makes the implementation of security algorithms particularly challenging.

This paper proposes a security framework for the generation of true random numbers, which are paramount as the core building block for many security algorithms; the intrinsic nature of wireless sensor nodes, and their capability of reliably providing measurements of environmental quantities, makes them natural candidates as true random number generators. In order to provide robustness to common attacks, we additionally devised a protocol aimed at obscuring the actual source of data, by making nodes cooperate with their neighbors. Furthermore, we describe an enhanced version of our framework consisting in an optimization for use in the context of resource-constrained systems.

1 Introduction

In the last few years Wireless Sensor Networks (WSNs) have become a widely used technology in several ICT application fields [9, 2]. As is well known, those networks are made up of a large number of small nodes that sense the environment and typically report their measurements to a base station; because of the need for protection against unauthorized access, trustworthiness of transmitted messages, and data integrity in general, security issues are of paramount importance and cannot be neglected when developing WSNs applications.

Commonly implemented security mechanisms rely on the availability of random numbers in order to perform their operations, as in the case of key exchange algorithms [7], which are based on randomly generated keys, or of mutual authentication algorithms [8, 5], which use the so called random "nonce" to ensure the other part is trusted. Random number generators therefore play a crucial role when considering security in ICT systems, and for wireless sensor networks in particular.

The main challenge when using random number sequences is the characterization of their statistical properties; typical requirements for such sequences are that they must not present any order nor any coherence, i.e. no regularity patterns should be discovered in a sequence.

Random numbers are typically generated either via *computational* or *physical* approaches. The *computational* approach relies on the use of formulae where a sequence of random numbers is generated as a function of a secret key and some initialization, the so-called random seeds; such generators are called Pseudo-Random Number Generators (PRNGs), since randomness is only apparent, as the produced sequences show some periodicity and are actually reproducible. The attacker could in principle collect generated random numbers until the sequence restarts, and then deterministically predict the rest of the sequence; a common used countermeasure consists in periodically changing the key after a certain time interval. PRNGs are widely used since they ensure good statistical properties and high bit rates, however their viability completely depends on the actual randomness of the initialization seeds.

As the name suggests, the *physical* approach relies on measurements related to physical phenomena; in this case, random numbers are generated as a function of a set of samples coming from sensory readings and the generator

is named a True Random Number Generator (TRNG), because the sequence is actually non deterministic and unpredictable. TRNGs do not need seeds nor secret keys, and do not exhibit periodicity because of the independence of the current random number from its past values. TRNGs usually require post-processing operations since they often do not ensure sufficiently good statistical properties, so they are intrinsically characterized by lower bitrates with respect to the PRNGs counterpart. TRNGs are thus often used as random seed generators for PRNGs since they are too slow for standalone usage in applications requiring high-bitrate random sequences.

However secure applications in the specific context of WSNs seldom require high-bitrate random sequences; we will thus focus on TRNGs as the basic building block for a security infrastructure for WSNs.

Moreover sensor nodes are obvious candidates as data providers for TRNGs, since their natural purpose is to collect great amount of data for environmental monitoring. As reported in the recent literature, many of the typically sensed quantities provide the necessary characteristics that a TRNG should possess, so in principle any sensor node could feed its own measurements to an on-board TRNG module; however, security issues make such choice impractical since the sensor board could be hacked by an attacker. A more suitable alternative is to rely on data from multiple sources, such as those originating from neighboring sensor nodes.

The contribution of this paper is the proposal of a TRNG module combined with a lightweight protocol for gathering multiple readings from nearby sensor nodes. Collected data are combined at the requester in order to generate the random number. As our considerations and experimental results will show, our solution is robust to security threats, since an attacker would have to tamper with multiple nodes before gaining complete control of the random number generation process, but at the same time it does not burden nodes with excessive computational or transmission requirements.

The remainder of the paper is structured as follows: Section 2 surveys the TRNGs presented in literature and their main features; our proposals for customizing TRNGs for use in resource-constrained devices are presented in Section 3, with additional details and a formal description of the communication protocols in Appendix A. A theoretical and experimental assessment of the proposed infrastructure is contained in Section 4, while Section 5 provides a comparison between our approach and other methods in the literature. Finally, Section 6 presents our conclusions.

2 Related Work

The construction of TRNGs has been widely discussed in literature, but most proposals have a narrow focus on the description of the physical process to be used to generate random numbers. For instance, the generator proposed by [12] used the small variations in the response time of the readings of a hard disk sector as the measurements source; in particular the authors observed that the rotation speed of a disk drive is not predictable and exhibits random fluctuations. In [14] the thermal noise present in resistors was proposed as source of randomness; the mean voltage sensed by the resistors is subtracted, and the noise component is then isolated. The work proposed by [26] built a TRNG by using the intrinsic features of quantum mechanics; in particular, the authors generate random bits by using photons emitted by light rays; the arrival time of the photon is used to evaluate which path was followed, thus producing a random bit. Many other works in literature show that, in general, physical quantities are actually suitable to build TRNGs [6, 28].

TRNGs specifically developed for Wireless Sensor Networks use the sensory component, the transceiver, or the internal clock as the main source of randomness. In [15], a function of the transmission power, namely the Received Strength Signal Indicator (RSSI), is used, while the authors of [11] exploit the noise contained in environmental readings. Another recent work [10] proposed a random number generator using the unpredictability of errors on transmission bits to generate numbers. Authors of [24] proposed a RNG that combines a clock and the CRC contained in the transmission packets.

As regards security, research on TRNGs has recently been shifting its focus toward the design of mechanisms to ensure that random numbers cannot be manipulated by an external attacker. In fact, regardless of the type of physical quantity chosen as randomness source, the generation process itself must be secure in its own implementation. Although all previous approaches showed that random numbers appear to be unpredictable for external attackers, they would still be able to violate the generator by physically breaching into the measurement system.

Physical attacks are usually performed to gain control of a sensor node. The work [4] shows that the so called "node capture", which effectively gives to the attacker full control of the sensor node is not very easy to be performed; however the same work claims that violating single components, and in particular the sensory one, is quite trivial. The *decentralization* of the measurements generation task has been proposed as a defense strategy in recent literature. Consider as example the work [3] where a P2P system is used to generate random numbers. Every peer p is allowed to generate a random number x_p; the network peers will agree on a true random number computed as $\oplus_p x_p$. In

this case it is proved that if at least $2m/3$ peers are not corrupted then the random number is correctly generated. The work [27] built up a TRNG using the phase jitter of the signal produced by a network of ring oscillators. This method too is tolerant to invasive and non-invasive attacks using some kind of redundancy in the transmission of the random bit strings among oscillators.

Although decentralization of the sensing task is helpful to build a secure TRNG, it is still subjected to security attacks because of the need for transmitting data. There is strong evidence in literature [30, 16, 22, 31] that physical attacks, and in particular "sensor tampering" are not very easy to address; at the same time, the MAC and network layers seem to be easier to protect using straightforward techniques like traffic padding, symmetric encryption and so on. The key property of using decentralization is that security issues are shifted from physical (challenging to secure) to upper layers (easier to secure).

3 A TRNG for Wireless Sensor Nodes

Nodes of a wireless sensor network need to rely on random numbers for all kinds of security purposes. Such nodes are typically severely constrained as regards their available resources, both in terms of storage capacity and of energy supply; however they are naturally able to sense common physical quantities, such as ambient temperature or humidity, battery voltage, and so on, which makes them suitable as true random numbers generators without the addition of expensive dedicated hardware.

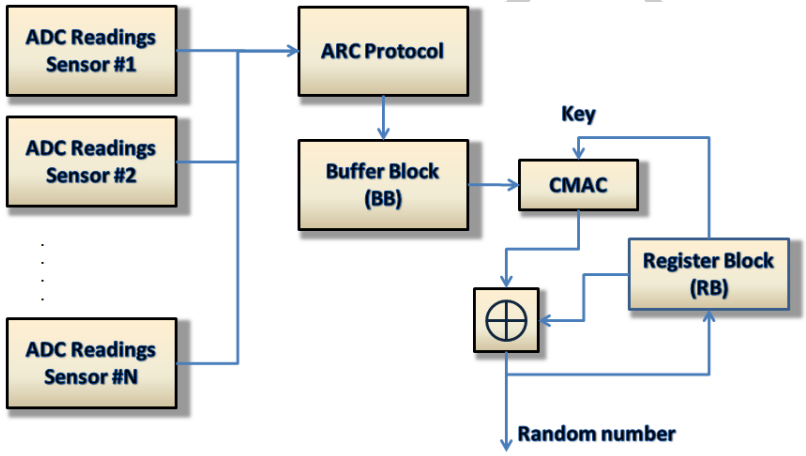


Figure 1: Block diagram for *ScatterRNG*.

The operation of a TRNG may be described in mathematical terms as follows:

$$r_t = f(m_{t-w}, m_{t-w+1}, ..., m_t), \tag{1}$$

where m_t is the reading related to the considered physical phenomenon at time t, w represents the size of the considered time window and r_t represents the output of the RNG.

A basic version of a random number generator for wireless sensor networks was proposed in [11], and consisted of a module generating truly random sequences starting from ADC readings produced from the node's sensor board. The ADC of a sensor node is used to sense the environment and its readings are sent to a 64-bit left-shift circular Buffer Block (BB); such readings are then encrypted by the CMAC algorithm in order to provide confusion and diffusion; finally, its 64-bit result is XOR-ed with a 64-bit Register Block (RB) to produce the random number as the output of the entire module. The register block also acts as a key for the CMAC, so the same set of readings will in general produce different outputs. The computation of the output random number RN may thus be formulated as:

$$RN = \text{CMAC}(BB, RB) \oplus RB, \tag{2}$$

where BB is the content of the Buffer Block, and RB is the content of the Register Block.

We note, however, that this generation process may be prone to attacks: measurements could be fraudulently obtained by physically violating the ADC block of the wireless sensor node, so it would be sufficient for an attacker to capture the RNG output for time t (i.e the secret key of the CMAC block at time $t+1$) to infer any subsequently generated random number.

In order to improve the original TRNG module, our recent proposal [18] suggested that the task of providing source data for the random number generator were shared among multiple nodes, so as to hide the actual source of data from potential attackers. The random number generation process was made to depend on multiple sources via an authenticated readings collection protocol, so that the attacker would be forced to tamper with a higher number of sensor boards before being able to gather a sufficient amount of information and break the TRNG.

We now discuss a further extension that provides an optimization for resource constrained devices, such as sensor nodes. Similarly to the original proposal, we rely on an authenticated readings collection protocol initiated by a requester, but we focus on reducing the required bandwidth and the overall message exchange in order to meet the tight resource constraints.

Since the RNG data source is scattered among many of the requester's neighboring nodes, for the sake of clarity we will refer to the original proposal as *ScatterRNG*, whereas our current improvement will be called $ScatterRNG_{light}$; the core of both algorithms is the Authenticated Readings Collection (ARC) protocol coupled to the TRNG module.

In the following, we describe *ScatterRNG*, with special focus on its ARC protocol, and the novel addition obtained by combining an enhanced version of the original TRNG module with an ARC_{light} protocol.

3.1 *ScatterRNG*

Our original proposal for a secure TRNG consisted in the addition of an authenticated readings collection (ARC) protocol to the basic core TRNG module, as depicted in Figure 1. The ARC protocol is in its turn composed of two main phases: *request dissemination* and *readings collection*. The aim of the dissemination phase is to notify the request for fresh ADC readings to the nodes within a k-neighborhood of the requester; such k-neighborhood is built via a greedy tree construction algorithm. During the collection phase, each of the neighbors will send a set of readings to its parent, which will select one of such sets out of all the ones it received from its children (including the one generated by itself), and repeat the same operation. The process terminates when the selected readings are delivered back to the requester, so that the data can be used as input to its TRNG module.

The format for messages of both phases of the ARC protocol is shown in Figure 2, where the fields have the following meaning:

- `type`: can take either of two different values: `request` or `reply`;
- `requester`: the identifier of the node initiating the dissemination phase;
- `sender`: the identifier of node actually sending the current message;
- `payload`: contains ADC readings when `type` is set to `reply`; it is unused for `request` messages;
- `TTL`: the number of remaining hops before reaching the leaf nodes in the neighborhood;
- `subtree cardinality`: in `reply` messages, it indicates the number of nodes belonging to the tree rooted at the `sender`; it is unused for `request` messages;
- `nonce`: used to ensure authentication;
- `hashcode`: used to ensure integrity; computed as the 8 least significant bytes of the SHA-1 digest of previous fields.

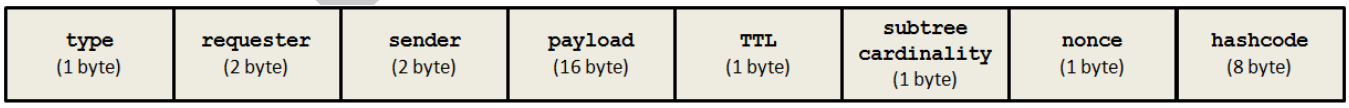


Figure 2: Fields of the ARC message.

In order to provide confidentiality for the ARC messages, we choose here to encrypt them according to CBC mode of operation of the DES using a 64-bit shared symmetric key, externally set by a pairwise key management scheme, as discussed in Section 4.3. The protocol also provides authentication and integrity by the combined use of the `hashcode` and `nonce`; in particular the nonce value is increased by 1 for reply messages, following the typical challenge-response pattern.

Finally, to ensure timely termination of the protocol, any node (including the requester) relies on a timer representing the maximum allowed round-trip-time for a request, computed as:

$$T = 2 \cdot TTL \cdot d_{max}, \tag{3}$$

where d_{max} is a predefined time constant indicating the maximum allowable single hop delay.

3.1.1 Request dissemination.

This phase is basically a flooding algorithm with limited scope, with the additional capability of constructing a tree over the k-neighborhood of the requester. The requester encrypts the initial message with the shared keys and sends it to its direct neighbors. Each receiver node decrypts the message and checks its validity against the hashcode, simply discarding it if not genuine. Upon passing the validity checks, the receiver stores the current values of the `requester`, `sender` and `nonce` fields; the sender identifier will be assumed as the parent of the tree built for the requester node, while the requester identifier will allow to prevent loops by rejecting any subsequent request message with the same requester. The receiver will forward a new encrypted ARC request message to all of its neighbors but the sender, only if `TTL` is greater than zero; before encrypting such request, the node will set itself as the sender, decrease the TTL, recompute the hashcode and then start the timer.

3.1.2 Readings collection.

An expired `TTL` triggers the beginning of the collection of readings.

Any leaf node creates a reply message by encapsulating its ADC readings in the `payload`, setting the `subtree cardinality` to 1, and modifying the `nonce` value to prevent replay attacks, as we previously described; the encrypted message is finally forwarded to the previously set parent node.

Any non-leaf node collects reply messages from its children until the timer T expires, and adds its own readings to the received ones; finally, it randomly selects one set of readings to be forwarded to its parent node. The probability according to which any of the nodes is selected as the provider of the readings is set to be proportional to the cardinality of the subtree rooted at that node.

Letting $n(\cdot)$ denote the function computing the cardinality of a subtree given its root node, and considering a sender node j, whose children set is $C(j)$, then the probability that readings coming from node z will be forwarded to the parent node is computed as:

$$p_{sel}(z) = \begin{cases} \frac{n(z)}{n(j)}, & \forall z \in C(j) \\ \frac{1}{n(j)}, & \text{if } z = j \end{cases} \tag{4}$$

As will be proved in Section 4, such selection rule ensures that the probability of selecting any of the reading sets is uniform, and equal to $1/N$ where N is the total number of nodes in the k-neighborhood; this means that the actual source of data is obscured to the potential attacker, which would be thus forced to tamper with all of the neighboring nodes to break the algorithm, if able to do so.

ARC messages during the collection phase are updated by the replier nodes by setting the `sender` value to their own id's, the `payload` to the chosen reading set, by updating the `subtree cardinality`, by setting `nonce` to the value of the previously received one plus 1, and by recomputing the `hashcode` via the SHA-1 function; finally, the reply message is encrypted.

3.2 *ScatterRNG*$_{light}$

ScatterRNG$_{light}$ is a lightweight version of the security framework described above and is specifically addressed to resource-constrained devices, such as wireless sensor nodes. The improvement consists in the use of the ARC$_{light}$ protocol together with an enhanced version of the basic TRNG.

It is well known that, for wireless sensor networks, sending/receiving a bit via radio transmission corresponds to about a few thousands of program instructions in terms of energy consumption [1], so the general guideline consists in trading communication for computation. In our case, the only software module involving radio transmissions is the ARC protocol, so we chose to reduce its complexity at the expenses of the TRNG module, which is completely computation-bound, thus reducing the overall energy requirements.

The ARC$_{light}$ protocol is tightly related to its forerunner, but we now limit the scope for choosing the source of data to the node's direct neighbors (i.e. the *1*-neighborhood, as opposed to the k-neighborhood of the previous

type (1 bit)	payload length (3 bit)	sender (12 bit)	payload (0-48 bit)	nonce (8 bit)	hashcode (56 bit)

Figure 3: Fields of the ARC_{light} message.

version). The random selection process is no longer needed, and the requester now simply concatenates all of the received reading sets before forwarding them to the RNG module, instead of choosing only one.

ARC_{light} is a bit-oriented protocol and its message format is depicted in Figure 3. It strikingly differs from ARC message in that the `requester`, `TTL` and `subtree cardinality` fields have been suppressed; the newly added `payload length` field takes on two different meanings for `request` and `reply` messages: it indicates the number of octets the requester wishes to receive for the former type, or the exact number of octets contained within the `payload` field for the latter. The `payload` field is entirely unnecessary and is suppressed for `request` messages. Unlike *ScatterRNG*, messages are no longer encrypted, but just authenticated: only the pair (`nonce` $\|$ `hashcode`) is encrypted using the CBC DES; the total size of those fields is set to 64 bits, as required by DES, so we accordingly set the length of `hashcode` to be 56 bits (whereas it was previously set to 64).

With respect to the previous version, the timer T has been modified to be adaptive, and is computed with the same formula used for the well-known TCP roundtrip time estimate: such choice provides adaptability to potential changes in the neighborhood topology, as is common in the context of wireless sensor networks. The value of T is guaranteed not to diverge, by thresholding it to a maximum value T_{max}, and is computed as follows:

$$T = \min\{RTT + 4 \cdot Dev, T_{max}\}, \tag{5}$$

where $RTT = (1-\alpha) \cdot RTT + \alpha \cdot RecRTT$ is the current roundtrip time estimate, $RecRTT$ is the roundtrip of the last received reply message, $Dev = (1-\beta) \cdot Dev + \beta \cdot |RecRTT - RTT|$ is the variation of RTT and α , β weigh the importance given to the past RTT and Dev values.

3.2.1 The Enhanced RNG module.

The basic RNG module has been enhanced in $ScatterRNG_{light}$ by adding a Hash Block, a Local Clock (LC) and a XOR operator, as depicted in Figure 4. The motivation behind the addition of the hash block is that the physical random source could provide *biased* input to our system (i.e. an unbalanced amount of 0's and 1's), so a practical deskewing technique is to use a Hash function as suggested in [20]; moreover we inserted the Local Clock in the random generation process to prevent an attacker with complete control of the sensory input from turning the module into a periodical RNG.

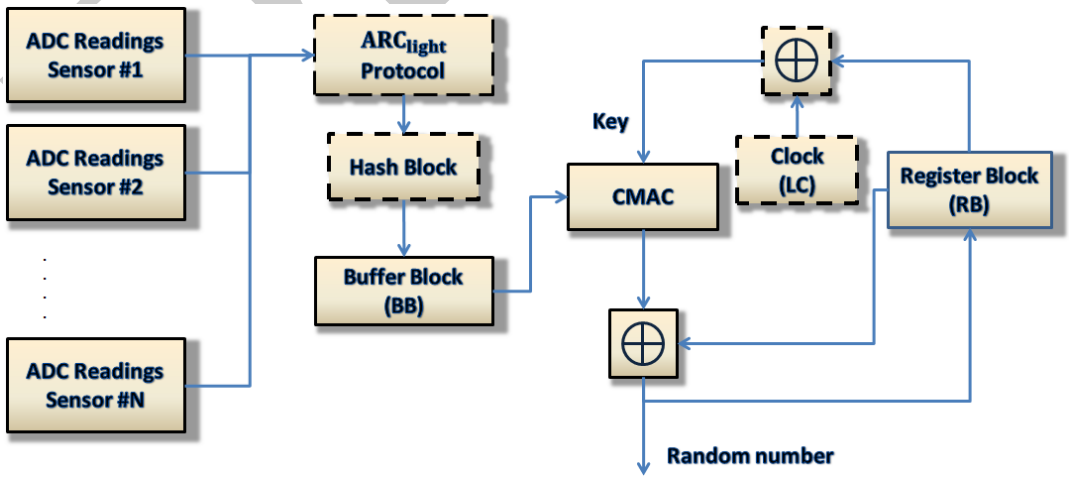


Figure 4: Block diagram for $ScatterRNG_{light}$, where dashed blocks represent the new additions to the previous proposal.

The enhanced RNG module accepts as input the concatenation of the reading sets X_i provided by the ARC$_{light}$ protocol and computes their digest (160 bits) via the SHA-1 function: Y =SHA-1$(X_1 \parallel X_2 \parallel \ldots \parallel X_n)$.

The Buffer Block (256 bits) is updated as follows:

$$BB = (BB << 160) + Y, \tag{6}$$

where << represents the bitwise left-shift operator.

The outcome of the Buffer Block represents the input for the CMAC algorithm, and the 64-bit result is, in its turn, XOR-ed with the content of a 64-bit register block (RB) to generate the true random number RN. The key of the CMAC is computed as $RB \oplus LC$, where LC is the least significant 16-bit value of a Local Clock (LC), with μ-second precision.

In summary, the random number is generated as follows:

$$RN = \text{CMAC}(BB, RB \oplus LC) \oplus RB. \tag{7}$$

4 Theoretical and Experimental Assessment

In order to prove the validity of our approach and its feasibility in a real scenario with resource-constrained distributed nodes, we assess the implementation of both versions of the ARC protocol in terms of their message and computational complexity; moreover, we will prove that the source data from the random numbers are uniformly scattered over the chosen neighborhood, i.e. the dependency of the generated random sequences is equally spread across the neighboring nodes, thus increasing robustness to attacks. We also provide a detailed explanation of the main security features of our proposal and finally we report the results of the experiments we conducted for testing the randomness of the generated sequences by using the NIST Test suite.

4.1 Message and Computational Complexity

The complexity of the ARC protocol can be analyzed by considering its two main phases: *request dissemination* and *readings collection*. Since nodes at each level of the tree need to send messages both to their parent and children nodes, the time complexity for both phases is $O(k)$, where k is the depth of the neighborhood. The message complexity is instead $O(n)$, where n is the number of nodes within the k-neighborhood, since each node (but the leaves) sends a request and each node (but the root) sends a reply.

As regards ARC$_{light}$, the time complexity is constant and $O(1)$ since only a *1*-neighborhood will be chosen to receive node readings; the message complexity is still $O(n)$, as for the previous version of the algorithm; it is worth noting however that, for all practical purposes, the amount of nodes in the considered *1*-neighborhood will be typically much lower than in the previous case.

From the viewpoint of energy efficiency, a comparison of the message format of the two protocols (see Figures 2 and 3) makes it clear that ARC$_{light}$ outperforms the original ARC protocol by at least a factor 2: the ARC message has a fixed length of 256 bits while its lightweight counterpart ranges from 80 up to 128 bits, for request and reply messages respectively. This means that the ARC_{light} protocol cuts the radio consumption at least to a half as compared to the previous version; moreover, the computational burden of transmitting ARC_{light} messages is reduced by a factor of 4 since the CBC mode of operation of DES needs to encrypt only one 64-bit block (`nonce` $\parallel$ `hashcode`) instead of four.

4.2 Uniform Scattering

The *ScatterRNG* infrastructure generates the random sequences by ensuring that the reading sets coming from a k-neighborhood of the requester are all chosen with equal probability.

As previously mentioned, every node applies the selection rule described by Eq. 4. Now, suppose that a specific reading set originating at node i is forwarded across the levels of the k-neighborhood and passes through nodes $i_1, i_2, .., i_t$ until it arrives at the requester r; by applying the product rule of probability, we obtain that the probability of such reading set to be chosen by the requester is:

$$p_{choice}(i) = \frac{1}{n(i)} \cdot \frac{n(i)}{n(i_1)} \cdot \frac{n(i_1)}{n(i_2)} \cdot \ldots \cdot \frac{n(i_t)}{N} = \frac{1}{N}. \tag{8}$$

This is sufficient to prove that the chosen data source for the RNG module is uniformly scattered with probability of $1/N$, given that N is the number of nodes within the requester's k-neighborhood.

Table 1: Threat model for the attacker.

Attack	Description
Sniffing	observes data exchanged among nodes
Replay	sends old and trusted information
Greyhole	forces sensor node to drop some packets
Blackhole	forces network traffic to pass through a specific sensor node, which will drop all packets
Radio tampering	disrupts the nodes' communications
Sybil attack	places malicious (Sybil) nodes next to legitimate ones
ACK spoofing	forges ACK messages to trick the sender into believing that a dead node is alive
Sensor board tampering	reads/corrupts the output of the ADC of a sensor board

The $ScatterRNG_{light}$ implementation restricts the depth of the neighborhood to 1, and avoids the use of random sampling during the selection of the input reading set; the requester just concatenates the received reading sets with the one coming from its own sensor board, and then processes them via the SHA-1 hash function. Since an inbuilt property of all secure hash functions is to make the digest equally dependent on all the input bits of the message, this is sufficient to prove that it is true also for the case at hand that all the nodes belonging to the 1-neighborhood (including the requester) will equally contribute to the random sequence.

4.3 Security Analysis

For the purpose of assessing our proposal from the standpoint of security, we provide here a detailed explanation of its main features. In order to customize the implementation of our TRNG for the context of wireless sensor networks we followed the guidelines reported in [30], and assumed a formal threat model (i.e. a description of the capabilities of the attacker) which is summarized in Table 1. The threat model will now be used to show how our infrastructure is capable of resisting to individual attacks or combinations of them.

Both *ScatterRNG* and $ScatterRNG_{light}$ assume the presence of a particular key distribution scheme, where sensor nodes are supposed to share pairwise symmetric keys to encrypt/authenticate their transmissions. A management scheme relying on a master shared symmetric key is not viable since, as soon as the key is stolen from a single node all network communications would be revealed, so that forging trusted information would become trivial for the attacker. Unfortunately, public key cryptography schemes are unpractical for WSNs, since they require large amounts of memory and computation, thus unacceptably shortening the overall network lifetime. According to current literature, the most practical approach is the use of a transitory master key scheme where sensor nodes are pre-set by node programmers with a master symmetric key, that in its turn allows for the distribution of pairwise shared keys. The master key is "transitory" in that it is destroyed by sensor nodes after the distribution phase to prevent a node-capture attack. Although the master key is removed from all nodes, this does not force the network to be static, provided that newly added nodes own the original master key: this prevents an untrusted attacker from joining the network. The computational burden due to the key distribution phase is sustainable by sensor nodes as the process only needs to be performed once at the beginning. Detailed description of such schemes is out of the scope of our proposal, but the interested reader can find all the technical details in [13, 25, 32, 17].

The security goals of *ScatterRNG* and $ScatterRNG_{light}$ are: (i) confidentiality, (ii) authenticity and (iii) integrity.

ScatterRNG achieves confidentiality by using the DES CBC mode of operation; originally, this was intended to prevent the attacker from guessing the content of the reading set to be used as the RNG data source. On the other hand, $ScatterRNG_{light}$ does not actually require to encrypt the incoming readings, since the order in which readings are received (or indeed whether they are received at all) is not predictable, which makes the problem of guessing the correct input to the RNG module computationally unfeasible for the potential attacker. Let us suppose the *1*-neighborhood is composed of N nodes, then the different ways the reading sets can be supplied to the RNG module amount to:

$$\sum_{j=1}^{N} \binom{N}{j} \cdot j! = \sum_{j=1}^{N} \frac{N!}{(N-j)!}, \tag{9}$$

where j is the number of received reading sets ($1 \leq j \leq N$), $\binom{N}{j}$ is the number of combinations of N elements out of j, and $j!$ is the number of permutations for the received reading sets. We therefore chose not to provide

confidentiality since, for sufficiently high values of N ($N \geq 21$), the system is no less robust than its encrypted counterpart.

Both versions of the protocol make use of a 1-byte nonce n to guarantee that reply messages are not a replay of old ones (i.e. authenticity). Any node receiving a request will update the nonce field with $nonce_{new} = nonce_{old} + 1$ to let the sender node know that the previous value of n was correctly decrypted. Finally, integrity checks are provided by using the `hashcode` field; in particular, the receiver node only needs to compare the decrypted hash value with the recomputed one, thus discarding the message if they are not equal. $ScatterRNG_{light}$ uses a shorter hash value than $ScatterRNG$; security is however not hindered because the attacker would on average need to generate as much as $2^{56}/2$ messages to forge a valid one.

We now provide an overview of the threats the attacker can devise against the secure random number generator, and the adopted countermeasures.

Sniffing. The attacker might try to infer some information by observing messages containing the readings received by the requester sensor node. As regards $ScatterRNG$, such attack would be completely ineffective since the node-to-node communications are encrypted. The second version, instead, just provides message authentication, so the attacker could in fact observe the readings while they are received at the requester node; however since the generation process still depends on the order of the received messages, the register block value, the data sensed by the requester node, and finally on the clock value, such attack would not provide sufficient information to guess the generated random number.

Replay. Trusted messages, previously observed using the sniffing attack, might be sent as fresh ones, thus "freezing" the output of the RNG module which would appear to be continuously using the same readings. Both versions of the RNG are secured against such type of attack thanks to the use of nonces; in particular, replayed messages would be discarded because the encrypted nonce would not agree with those sent by the requester node. The probability the attacker sends a trusted message is $1/2^8$, given the size of the `nonce` field.

Greyhole/Blackhole. The attacker might try to isolate a specific sensor node (the requester) from the rest of the network thus lowering the randomness of the generated sequence. $ScatterRNG$ is sensitive to such type of threats, as the attacker could force the requester to use only the requester's own readings as sources, thus breaking the random source selection algorithm, and potentially the whole system, if the attacker can additionally breach into the sensor board. $ScatterRNG_{light}$ prevents the attacker from gaining control of the generation process since we had the random numbers depend also on the internal clock value.

Radio tampering. The attacker might tamper with the radio component of the requester, thus causing the failure to receive reply messages from neighboring nodes, which would result into an indefinite wait for the requester node. Both versions of the algorithm solve the issue by using a timer that forcibly triggers the production of the node's own readings to keep the process going. The attacker can only slow down the generation process, since the value of T will rapidly reach the upper bound set by the node programmer. (see Eq. 3 and 5).

Sybil attack. The attacker could resort to place a malicious node near some other sensor node and try to trick the receiver into believing that is a trusted node. $ScatterRNG$ is robust against such type of attack thanks to the use of encryption; only trusted nodes knows the pairwise keys so any message sent by a Sybil node would be discarded by the requester node with a probability: $p_{discard} = 1 - 1/2^{64}$. $ScatterRNG_{light}$ uses a lightweight solution relying on authentication to solve the issue. The attacker's probability of generating a trusted message is still $p = 1/2^{64}$, as the `hashcode` and `nonce` fields together amount to 64 bit. The Sybil attack differs from a replay attack only for the rate of authenticated messages sent by the malicious user; similarly to what happens during a replay attack, a flood of authenticated messages is ineffective since the randomness of the generated sequence is not substantially altered.

ACK spoofing. The attacker aims to make the requester node believe that a dead node is instead still alive. Both versions of the algorithm were protected by such attack by the use of the nonce technique.

Sensor board tampering. The attacker might read the content of the sensor board or directly tamper with the readings from the ADC, forcing them to be set to specific values. Both $ScatterRNG$ and $ScatterRNG_{light}$ behave quite similarly by making use of the readings coming from the requester's neighborhood just to obscure the actual data source to the attacker. The attacker would thus be forced to violate all the neighboring sensor boards before they can entirely control the selection of the readings to be used as input to the RNG: as a consequence $ScatterRNG$ would turn into a PRNG, and the generated sequence would simply be periodic; on the other hand, $ScatterRNG_{light}$ would continue to exhibit aperiodicity thanks to the use of the local clock. However

Table 2: NIST test suite.

Test	Description
Frequency	tests whether the same number of zero and one appears in the sequence.
Block frequency	quite similar to the previous test, but differs from it because the test is performed within blocks of fixed length
Runs	computes the number of runs of zero and one for different lengths
Longest run	tests whether the longest run of ones is not longer than a specific threshold for a given sequence length
Binary matrix rank	the sequence is partitioned into submatrices whose rank should be full. Such test is used to check the existence of linear dependencies among chunks of the sequence
Discrete Fourier Transform	peaks in the frequency domain indicate some kind of periodicity within the random sequence
Non-Overlapping template matching	counts the number of occurrences of a specific pattern within the random sequence. Whenever the pattern is found, the search continues after the last bit of the sequence that matched with the pattern
Overlapping template matching	quite similar to the previous test, however when the target pattern is found, the search will continue by shifting only one bit within the random sequence
Maurer's universal statistical	checks whether the sequence is compressible without loss of information. Compressibility of the sequence would indicate some kind of non-randomness
Linear complexity	the focus of the test is to compute the length of a linear feedback shift register (LFSR) that reflects the complexity of the sequence. If the length of LFSR is too low with respect to the length of the sequence, then the sequence is non-random
Serial	tests whether all possible m-bit patterns are equally distributed within the random sequence
Approximate entropy	quite similar to the previous test but its aim is to check the frequency of pairs of overlapping blocks of consecutive length (m and $m+1$)
Cumulative sums	computes the sum of a biased sequence $(-1, 1)$ that should be near zero for all the length of the sequence
Random excursion	computes the cumulative sums and checks if particular states (+4,+3,...,-3,-4) are visited with the distribution probability that should have a random sequence
Random excursion variants	the same as the previous test, with the difference that the states that will be analyzed vary among a wider range (+9,+8,....,-8,-9)

we want to point out out that the 16-bits local clock, when taken as the single source of randomness, cannot guarantee the same quality of a generator that is not under attack. The issue is discussed in detail in Section 5.

Combined sensor board + radio + clock tampering. The attacker could prevent any new readings from being delivered to the RNG module. In this case, both versions of the algorithm would collapse into a cryptographic pseudo-random number generator. The only source of randomness would be provided by the register block, which is however dependent on its past values, and the generator would be completely compromised when the sequence restarts. Such suite of attacks, when enacted at once, is usually known as the "node capture" attack that was proved to be hard to counteract by [4]; also [13] has shown that the attack is viable using common devices, such as a laptop, a programming board and JTAG debugging device. Whenever the node is physically captured there is no way to restore its proper working status; nevertheless, thanks to the use of the transitory master key scheme, the attacker will be unable to compromise the communications of the remaining network nodes, since the master key is no longer available after the key distribution phase.

4.4 Randomness Quality Assessment

We experimentally assessed the performance of both *ScatterRNG* and $ScatterRNG_{light}$ in terms of the randomness of the produced number sequences; namely we used the NIST Test suite to check that the generated sequences may be classified as truly random. Such suite implements 15 different tests allowing to check for the presence of statistical anomalies within a random sequence; a brief description of them is provided in Table 2.

Each statistical test takes a sequence of bits as input, and generates the so-called *P-value*, which indicates the probability for a perfect random number generator to produce a sequence that is less random than the tested one. To guarantee that any statistical anomaly of the random number generator is revealed, it is necessary to perform the tests on a large number of sequences (at least 55) in order to obtain a more informative histogram of the P-values. The P-values are usually plotted in the x-axis of the resulting histogram, which is divided into 10 equally spaced bins, while the y-axis represents the number of sequences belonging to that bin. The histogram is thus expected to be

Table 3: Network parameter settings for *ScatterRNG* and $ScatterRNG_{light}$

Parameter	**Setting for ScatterRNG**	**Setting for ScatterRNG$_{light}$**
Size of each reading (l)	8 bits	8 bits
Number of readings per message (p)	16	6
Number of nodes (N)	10	10
Hop delay (d_{max})	0.5 sec.	-
Maximum roundtrip time (T_{max})	-	1 sec.
Weight for RTT (α)	-	0.125
Weight for Dev (β)	-	0.25

nearly uniformly distributed, and its χ^2 index is computed as follows:

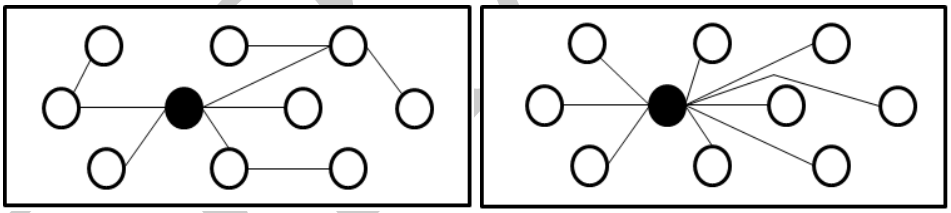
$$\chi^2 = \sum_{i=1}^{10} \frac{(F_i - s/10)^2}{s/10}, \tag{10}$$

where F_i is the number of P-values belonging to the i-th bin, and s is the number of tested sequences. It is then necessary to compute a new index defined as $P\text{-}value_T = igamc(9/2, \chi^2/2)$, where $igamc(\cdot,\cdot)$ is the "incomplete gamma function"; if such index is greater than 10^{-4} then the sequences may be considered uniformly distributed .

The uniform distribution of the P-values, however, is just a necessary condition to claim a random number generator as true. Another important index to be computed is the *ratio*, which indicates the amount of sequences with a P-value greater than a certain threshold θ. Typically, the value of the threshold is chosen to be $\theta \in [0.001, 0.01]$. For a given value of the threshold, the constraint on the minimum required ratio is the following:

$$ratio > p - 3\sqrt{\frac{p(1-p)}{s}}, \tag{11}$$

where $p = 1 - \theta$.



(a) (b)

Figure 5: The network topology for experiments: (a) *ScatterRNG* and (b) $ScatternRNG_{light}$; the requester node is the dark one.

We tested both *ScatterRNG* and $ScatterRNG_{light}$ on a network composed by 10 TelosB nodes deployed within a 25×15 m^2 area in our laboratory; the actual deployment is shown in Figure 5. We arranged nodes into a 2-neighborhood of the requester (dark node) for the purposes of *ScatterRNG*, whereas we imposed a star topology in the case of $ScatterRNG_{light}$. For conducting these experiments, we run both algorithms and collected readings from the sensor boards about the ambient light (8-bit) for a period of 10 consecutive days; as the RNG module at the requester generated the random numbers, we let it store them into a connected workstation, until 10 Mbits of data were accumulated. For the aim of validation, we finally partitioned such data into 1,000 sequences, each 10^4 bits long, and processed them using the NIST Test suite.

The network parameter settings for both sets of experiments are summarized in Table 3, and the parameter settings for the NIST tests are reported in Table 4. Table 5 shows the corresponding results in terms of $P\text{-}value_T$ and of the obtained success *ratio*. In both sets of experiments. the resulting values for $P\text{-}value_T$ and *ratio* are above the pre-set thresholds, which allows us to conclude that both security frameworks perform satisfactorily in terms of true randomness degree; moreover, $ScatterRNG_{light}$ is more efficient, for the reasons discussed earlier, thus proving overall preferable in the context of a resource-constrained scenario.

Table 4: Parameter settings for the NIST test suite

Parameter	**Value**
Length of each sequence (L)	10^4 bits
Number of tested sequences (s)	1000
Threshold for P-values (θ)	0.01
Ratio value (*ratio*)	0.9806

Table 5: Results for $P\text{-}value_T$ and *ratio* for the two versions of the RNG.

Test name	*ScatterRNG*		$ScatterRNG_{light}$	
	$P\text{-}value_T$	*ratio*	$P\text{-}value_T$	*ratio*
Frequency	0.9015	0.9840	0.1223	0.9830
Block Frequency	0.2133	0.9810	0.3508	0.9910
Runs	0.3512	0.9910	0.1223	0.9810
Longest Run	0.5341	0.9900	0.5341	0.9910
Binary matrix rank	0.5341	0.9920	0.7351	0.9850
Discrete Fourier Transform	0.0674	0.9890	0.2135	0.9910
Non overlapping template matching	0.2137	0.9960	0.4602	0.9890
Overlapping template matching	0.2120	0.9810	0.3509	0.9830
Maurer's universal statistical	0.8175	0.9810	0.8065	0.9990
Linear Complexity	0.9310	0.9990	0.8965	0.9920
Serial	0.3508	0.9840	0.5348	0.9970
Approximate entropy	0.9903	0.9950	0.7451	0.9950
Cumulative sums	0.3507	0.9930	0.7392	0.9880
Random excursion	0.6529	0.9880	0.6402	0.9810
Random excursion variants	0.7543	0.9890	0.7502	0.9940

5 Comparison against other TRNGs

The characteristics of our two proposals in terms of resilience are now compared against three different RNGs for wireless sensor networks using the threat model of table 1. We also discuss the behavior of both the original *ScatterRNG* and of its lightweight version when they are stripped off of the ARC/ARC_{light} communication protocols. Finally, we exclude node capture from the resilience analysis as we assume that the node is definitively under control of the attacker after that all of its components are tampered.

Table 6 shows the results of the comparison: for different threats, each cell shows whether the method indicated in the heading of the column is resilient to the threat (✓) or not (✗); an empty cell means that the threat is not applicable.

The first two columns contain the results for the versions of *ScatterRNG* and $ScatterRNG_{light}$ without the respective communication protocols: due to the absence of radio communications, the only viable attacks could be performed against the sensor or the clock components. As regards *ScatterRNG*, sensor tampering would turn it into a periodical RNG; on the other hand, $ScatterRNG_{light}$ is resilient to clock tampering and sensor tampering when performed individually. Indeed, under sensor tampering attack, we observed that the generated sequences do not pass the Overlapping template test (ratio is below the threshold) while the Maurer's Universal Statistical test shows a bad distribution of the P-values. In particular, as regards the template matching test, the results indicate that, sometimes, the attacker might be able to infer subsets of the generated outputs as the size of the templates is small as compared to the output (6-8 bits versus 64 bits). As regards the Maurer's Universal Statistical test, the distribution of P-values of the output sequences was very poor (all sequences have a P-value between 0.3 and 0.4); however the Ratio was over the threshold for all sequences. The use of the clock in $ScatterRNG_{light}$, anyway, avoids that sensor tampering turns it into a periodical RNG.

The third and fourth columns consider *ScatterRNG* and $ScatterRNG_{light}$ combined with ARC/ARC_{light} protocols: they were designed to be resilient to each of the threat we listed in Table 1, and as regards security issues they behave equivalently; however we proved that $ScatterRNG_{light}$ is better than *ScatterRNG* in terms of message and

Table 6: Comparison of different random number generators against common security threats.

Threat	$SRNG_{light}$ without ARC_{light}	$SRNG$ without ARC	$SRNG_{light}$	$SRNG$	Francillon et al. [10]	Latif et al. [15]	Rhee et al. [24]
Sniffing			✓	✓	✓	✓	✓
Replay			✓	✓	✓	✓	✗
Greyhole/Blackhole			✓	✓	✗	✗	✗
Sybil			✓	✓	✓	✓	✗
Ack spoofing			✓	✓	✓	✓	✓
Radio Tampering			✓	✓	✗	✗	✗
Sensor Tampering	✓(*)	✗	✓	✓			
Clock Tampering	✓		✓	✓			✗

(*) Under sensor tampering, the RNG failed to pass the Overlapping template matching and the Maurer's Universal Statistical tests.

computational complexity as well as the energy consumption, so the light version should be preferred to the basic one.

The remaining columns list three other approaches for RNG in WSNs. The proposal of Francillon et al. [10] generates random numbers by exploiting the bit errors of the communication packets; the authors proved that such bit errors are unpredictable using sniffing attacks even though the attacker is placed very near to the victim node. Their method is not resilient to BlackHole as the attacker could easily stop the reception of fresh bits (using Jamming), thus turning the RNG into a periodical one; the same consideration holds for the radio tampering attack. Although their proposal implements the CMAC and rekeying techniques, and adds the physical source as generator of randomness, the authors did not assess the output sequences using statistical tests, so we can only hypothesize about the true randomness of the method.

Latif et al. [15] propose a quite similar approach that relies on the RSSI of radio packets as source of randomness; the method assumes sensor nodes that move within the field so RSSI change over the time. RSSI is computed onboard thus sniffing would be uneffective; replay, sybil and ack spoofing attacks cannot lower the randomness of the sequences provided that the node continues to receive packets from other trusted nodes; on the contrary, their method is sensitive to Radio tampering and Blackhole as it uses only the radio component as source of randomness. Authors tested the single stream output using NIST Test Suite and successfully passed all the tests; the NIST software documentation [23] however specifies that in order to obtain statistically meaningful results at least 55 sequences should be analyzed, thus also in this case we cannot be completely confident about the quality of the generator.

The generator proposed by Rhee et al. [24] generates random sequences combining randomness coming from radio and clock components. They use a 8-bit clock as random seed generator and the 8-bit CRC of the received packets to implement rekeying. In principle, the method could be resilient to combined attacks as the randomness comes from different physical sources; however the work in [21] showed that such generator suffers of many design flaws. First of all, under clock tampering attack, the sequence restarts each three time steps, i.e. $RN(t) = RN(t + 3 * m)$ for any $m \in \mathbb{Z}$ and the generator becomes completely predictable; the authors propose to rekey the method every ten time steps, but a Blackhole attack, as well as radio tampering, force the generator to remain within its three values forever. The only attacks that appear to have no effect are sniffing and ack spoofing as they do not affect the internal state of the generator. Finally, the method was tested using the ENT batteries of tests [29] that clearly showed its non-randomness also when not under attack.

6 Conclusions

This work proposed a secure true random number generator in the context of Wireless Sensor Networks, relying on the sensed measurements as the source of randomness. Our TRNG may be effectively employed for multiple purposes, such as the generation of keys, nonces, random primes, and so on, which are all very relevant for the security of a WSN.

The RNG module might represent a critical part of the system since an attacker might focus on it to break the whole framework; this is why we implemented an Authenticated Readings Collection protocol to hide the actual data provider and eliminate the presence of a single weak point, thus improving the overall robustness.

We particularly focused on the optimization of transmissions, which usually cause the highest energy burden in WSNs, and developed a lightweight version of the protocol that significantly reduces the energy requirements while maintaining the same degree of randomness, as discussed in Section 4.

The use of a transitory master key scheme prevents network communications from being compromised by a node capture attack; at the same time, the use of symmetric key encryption keeps the computational burden limited as compared to public key schemes.

The randomness tests were conducted over many input sequences using the NIST Test Suite, and have shown that the method generates true random numbers; moreover the comparison with other recent methods clearly demonstrates that our proposal is the only one to be resilient to a large suite of attacks described in the formal threat model. As a further development, in order to gain additional robustness, we plan to allow the requester to choose which neighbors should be selected as potential sources, regardless of the network topology, always keeping the energy constraints in mind.

References

[1] G. Anastasi, M. Conti, M. Di Francesco, and A. Passarella. Energy Conservation in Wireless Sensor Networks: A Survey. *Ad Hoc Networks*, 7(3):537 – 568, 2009.

[2] G. Anastasi, G. Lo Re, and M. Ortolani. WSNs for Structural Health Monitoring of Historical Buildings. In *Proceedings of 2nd Conference on Human System Interactions*, pages 574 –579, 2009.

[3] B. Awerbuch and C. Scheideler. Robust Random Number Generation for Peer-to-Peer Systems. In *Principles of Distributed Systems*, pages 275–289. Springer Berlin Heidelberg, 2006.

[4] A. Becher, Z. Benenson, and M. Dornseif. Tampering with Motes: Real-world Physical Attacks on Wireless Sensor Networks. *Security in Pervasive Computing*, 3934:104–118, 2006.

[5] Z. Benenson, N. Gedicke, and O. Raivio. Realizing Robust User Authentication in Sensor Networks. In *Proceedings of the first Workshop on Real-World Wireless Sensor Networks*, 2005.

[6] S. Callegari, R. Rovatti, and G. Setti. Embeddable ADC-based True Random Number Generator for Cryptographic Applications Exploiting Nonlinear Signal Processing and Chaos. *IEEE Transactions on Signal Processing*, 53(2):793.805, 2005.

[7] S.A. Camtepe and B. Yener. Key Distribution Mechanisms for Wireless Sensor Networks: a Survey. Technical report, Rensselaer Polytechnic Institute, Computer Science Department, 2005.

[8] T.H. Chen and W.K. Shih. A Robust Mutual Authentication Protocol for Wireless Sensor Networks. *ETRI Journal*, 32(5):704–712, 2010.

[9] D. Estrin, L. Girod, G. Pottie, and M. Srivastava. Instrumenting the World with Wireless Sensor Networks. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 2033–2036, 2001.

[10] A. Francillon and C. Castelluccia. TinyRNG: A Cryptographic Random Number Generator for Wireless Sensors Network Nodes. In *Proceedings of the 5th International Symposium on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks and Workshops*, pages 1–7, 2007.

[11] V. Gaglio, A. De Paola, M. Ortolani, and G. Lo Re. A TRNG Exploiting Multi-Source Physical Data. In *Proceedings of the 6th ACM Workshop on QoS and Security for Wireless and Mobile Networks*, pages 82–89, 2010.

[12] M. Jakobsson, E. Shriver, B.K. Hillyer, and A. Juels. A practical Secure Physical Random Bit Generator. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 103–111, 1998.

[13] D. Jing, C. Hartung, R. Han, and S. Mishra. A Practical Study of Transitory Master Key Establishment For Wireless Sensor Networks. In *Proceedings of the First International Conference on Security and Privacy for Emerging Areas in Communications Networks (SecureComm) 2005*, pages 289–302, 2005.

[14] B. Jun and P. Kocher. The INTEL Random Number Generator, 1999.

[15] R. Latif and M. Hussain. Hardware-based Random Number Generation in Wireless Sensor Networks (WSNs). *Advances in Information Security and Assurance*, 5576:732–740, 2009.

[16] Z. Li and G. Gong. A Survey on Security in Wireless Sensor Networks. Technical report, Department of Electrical and Computer Engineering, University of Waterloo, Canada, 2011.

[17] C. H. Lim. LEAP++: A Robust Key Establishment Scheme for Wireless Sensor Networks. In *Proceedings of the 28th International Conference on Distributed Computing Systems Workshops*, pages 376–381, 2008.

[18] G. Lo Re, F. Milazzo, and M. Ortolani. Secure Random Number Generation in Wireless Sensor Networks. In *Proceedings of the 4th international conference on Security of information and networks*, pages 175–182, 2011.

[19] N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[20] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC press, 2010.

[21] P. Peris-Lopez, J.C. Hernandez-Castro, J.M. Tapiador, E. San Millán, and J.C. Van der Lubbe. Security Flaws in an Efficient Pseudo-Random Number Generator for Low-Power Environments. In *Security in Emerging Wireless Communication and Networking Systems*, pages 25–35. Springer, 2010.

[22] A. Perrig, J. Stankovic, and D. Wagner. Security in Wireless Sensor Networks. *Communications of the ACM*, 47(6):53–57, 2004.

[23] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. Technical report, NIST-National Institute of Standards and Technology, 2010.

[24] D. Seetharam and S. Rhee. An Efficient Pseudo Random Number Generator for Low-Power Sensor Networks. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 560–562, 2004.

[25] A.N. Shen, S. Guo, H.Y. Chien, and M. Guo. A Scalable Key Pre-Distribution Mechanism for Large-Scale Wireless Sensor Networks. *Concurrency and Computation: Practice and Experience*, 21(10):1373–1387, 2009.

[26] A. Stefanov, N. Gisin, O. Guinnard, L. Guinnard, and H. Zbinden. Optical Quantum Random Number Generator. *Journal of Modern Optics*, 47(4):595–598, 2000.

[27] B. Sunar, W.J. Martin, and D.R. Stinson. A Provably Secure True Random Number Generator with Built-In Tolerance to Active Attacks. *IEEE Transactions on Computers*, 56(1):109–119, 2007.

[28] I. Vasyltsov, E. Hambardzumyan, Y.S. Kim, and B. Karpinskyy. Fast Digital TRNG Based on Metastable Ring Oscillator. In *Cryptographic Hardware and Embedded Systems*, pages 164–180. Springer, 2008.

[29] J. Walker. Randomness Battery, http://www.fourmilab.ch/random/, 1998.

[30] Y. Yu, K. Li, W. Zhou, and P. Li. Trust Mechanisms in Wireless Sensor Networks: Attack Analysis and Countermeasures. *Journal of Network and Computer Applications*, 35(3):867 – 880, 2011.

[31] W. T. Zhu, F. Gao, and Y. Xiang. A Secure and Efficient Data Aggregation Scheme for Wireless Sensor Networks. *Concurrency and Computation: Practice and Experience*, 23(12):1414–1430, 2011.

[32] Zhu, S. and Setia, S. and Jajodia, S. LEAP+: Efficient Security Mechanisms for Large-Scale Distributed Sensor Networks. *ACM Transactions on Sensor Networks*, 2(4):500–528, 2006.

A ARC Automaton

In order to provide a formal and unambiguous description of the operations of ARC/ARC_{light} protocol, we will regard it as a distributed algorithm run concurrently by each network node i; loosely following the notation provided by [19], each node's ARC module will be seen as a process i running the same I/O automaton.

An I/O automaton $\mathcal{A}$ is formally defined by five components:

1. $sig(\mathcal{A})$, a set of actions involving the automaton;
2. $states(\mathcal{A})$, a set of states;
3. $start(\mathcal{A})$, a nonempty subset of $states(\mathcal{A})$ also known as initial states;
4. $trans(\mathcal{A})$, a relation of type $states(\mathcal{A}) \times sig(\mathcal{A}) \times states(\mathcal{A})$; for each state s and input action π, the transition $(s,\ \pi,\ s') \in trans(\mathcal{A})$ leads to the new state s';
5. $tasks(\mathcal{A})$, an equivalence relation on $local(sig(\mathcal{A}))$, where $local(sig(\mathcal{A}))$ represents the set of locally controlled actions; the automaton is also supposed to give fair turns to each task.

In this formalism, the actions performed by the automaton (i.e. its *signature*, $sig(\mathcal{A})$) act on the *state* variables and change their values by triggering the corresponding *transitions*. Actions may be classified as *input* actions, which are triggered by external processes, as opposed to *output*, *internal* and *external* ones, which are locally controlled by the automaton using *tasks*: they can be thought as thread of control and model the concurrent execution of internal operations within the automaton. It is worth noting that the *external* actions should be thought as third-party library functions; for simplicity's sake, we will omit their implementation details. Finally, whenever required, we will split up the automaton code for ARC (left column) and ARC_{light} (right column).

The complete signature for the ARC/ARC_{light} automaton, $\mathcal{A}_{ARC}$, is presented in Table 7.

In order to keep trace of individual ARC requests, it is useful to introduce the concept of *session*, defined as the following set of variables:

- *requester*: the id of the starter node;
- *parent*: the node from which the request is received (same as *requester* for ARC_{light});
- TTL: the number of remaining hops before reaching the leaf nodes in the neighborhood;
- T: a decreasing timer that forces the termination of the protocol if needed;
- *replies*: the reading sets received by the neighbors;
- *cardinalities*: the cardinality of subtrees rooted at neighbors (useful for ARC only);
- *nonce*: used to ensure authentication of reply messages.

In principle, multiple nodes might start the random number generation process at the same time and each node should be capable of managing multiple requests concurrently; we thus chose to group all the pending sessions within the state array S. In the following we will suppose S is indexed by the *requester* component or by a numerical index, so $S(z)$ indicates a particular session with $requester = z$ or the session at the position z in the array (the indexing type will be clear from the context). In addition we will use dot notation to extract a particular component from a given session (for instance $S(z).parent$ represents i's parent for the session started by z, or the i's parent for the z-th session). As regards the *Secret* array, its j-th component indicates the pairwise shared key between i and j. Table 8 reports the complete set of variables composing the $\mathcal{A}_{ARC}$ internal state. Note that the $Delay_{max}$ and *Desired* variables are externally set by the node programmer, while the *Secret* array is provided by the chosen pairwise keys management scheme.

A.1 Transitions

Here we report the most relevant transitions i.e.: *init*, *receive*, *decide* and *makeInput*. The remaining transitions are very easy to be derived and lack of interesting implementation details. The *init* transition starts the request dissemination protocol and accepts as input the TTL value. The light version must be invoked with $TTL = 1$. The precondition block ensures that node i is not running another request for random number generation for itself. The effect block creates a new session by setting the related parameters as in the scheme provided above. The value of the timer T is set by the *estimateRTT* function that uses Equation 3 or Equation 5 based on the running protocol. Clearly, if $TTL > 0$, node i sends a request message to its neighbor nodes $nbrs_i$.

$init(TTL)_i$

PRECONDITION:
foreach $j \mid 1 \leq j \leq length(S)$ **do**
assert($i \neq S(j).requester$);
EFFECT:
$Nonce := Clock$;
call $estimateRTT(TTL, Delay_{max})$;
add $(i, '', TTL, RTT, '', '', Nonce)$ **to** S ;
if $TTL > 0$ **then**
foreach $j \in nbrs_i$ **do**
call $makeRequest()$;
add $ToEncrypt$ **to** $send(j)$;

The second transition of the $\mathcal{A}_{ARC}$ automaton is *receive* which is triggered by the two input messages *request* (received during the dissemination phase) and *reply* (received during the readings collection phase).

Whenever i receives a message it immediately decrypts its content using the *decrypt* function. The fields of the decrypted message m are accessed still using the dot notation. If the received message is a *reply*, i firstly checks for its authenticity by comparing the recomputed hashcode against the received one, and then also controls if the message is not an old replay, by verifying that the received *nonce* is equal to those sent during request dissemination plus one. If the message is genuine i adds the *payload* and *cardinality* fields to the related session of ARC and only the *payload* field for ARC_{light}.

$receive(m)_{j,i}$

PRECONDITION:
call $decrypt(m, Secret(j))$;
$m.type =$ "reply";
$m.hashcode$=SHA-1(m) ;
$m.nonce$=$S(requester).nonce + 1$;
EFFECT:
add $m.payload$ **to** $S(m.requester).replies$;
add $m.cardinality$ **to** $S(m.requester).cardinalities$

$receive(m)_{j,i}$

PRECONDITION:
$m.type =$ "reply";
call $decrypt(m.nonce \| m.hashcode, Secret(j))$;
$m.hashcode$=SHA-1(m);
$m.nonce$=$S(i).nonce + 1$;
EFFECT:
add $m.payload$ **to** $S(i).replies$;

As regards the reception of a request message, the precondition block checks that $m.type = request$, recomputes the message hashcode and compares it against the received one. If the two values do not match, i discards the message considering it as not genuine. The $''$ symbol means that the considered field is filled with a $NULL$ value.

The effect block of the ARC version builds a tree, so i accepts a request message if and only if the *joined* variable signals that there is no other pending session for the same requester,. If the request is accepted, i creates a new session, and if it is not a leaf ($TTL > 0$) it also sends a request message to every neighbor but the parent j. In the opposite case, request message is simply discarded.

The ARC_{light} effect block is much more simple and only requires to set the RTT to zero and to set up the incoming session. In this case, no messages are sent to the immediate neighbors since the TTL is fixed to 1.

receive$(m)_{j,i}$

PRECONDITION:
call *decrypt*$(m, Secret(j))$;
$m.type$ ="*request*";
$m.hashcode$=SHA-1(m) ;
EFFECT:
$joined := false$;
foreach $z \mid 1 \leq z \leq length(S)$ **do**
  if $m.requester = S(z).requester$ **then**
    $joined := true$;
if $joined = false$ **then**
  call *estimateRTT*$(m.TTL, Delay_{max})$;
  add $(m.requester, j, m.TTL, RTT, '', '', m.nonce)$ **to** S;
  if $TTL > 0$ **then**
    foreach $z \in nbrs_i - \{j\}$ **do**
      call *makeRequest*();
      add *ToEncrypt* **to** *send*(z);

receive$(m)_{j,i}$

PRECONDITION:
$m.type$ ="*request*";
call *decrypt*$(m.nonce \| m.hashcode, Secret(j))$;
$m.hashcode$=SHA-1(m) ;
EFFECT:
$RTT := 0$;
add $(j, j, 0, RTT, '', '', m.nonce)$ **to** S;

Another important transition is *decide*; it is triggered whenever the timer T expires and behaves in two different ways, depending on whether i is a requester or not. If i is a requester, its output parameter is the *RdSet* variable, that is the reading set to be sent to the RNG module; otherwise, *RdSet* is forwarded into a *reply* message to i's parent.

decide$(RdSet, requester)_i$

PRECONDITION:
$S(requester).T \leq 0$;
EFFECT:
call *makeInput*$(S(requester).replies, S(requester).cardinalities)$;
if $i \neq requester$ **then**
  $j := S(requester).parent$;
  $nonce := S(requester).nonce + 1$;
  call *makeReply*$(requester)$;
  add *ToEncrypt* **to** *send*(j);
remove $S(requester)$ **from** S;

The *makeInput* action accepts as input the arrays $(replies, cardinalities)$ and computes the variables *RdSet* (to be sent to the RNG module) and *Cardinality* (*ARC* only). As regards *ARC*, the *RdSet* is chosen among the received sets of reading exploiting the selection rule described in Eq. 4. In order to include itself in the selection of *RdSet*, i also adds its *LocReadings* to the proposal list. Note that *LocReadings* is set by calling the external *sense*() function wired to the sensory components of the node. The for loop allows to compute the cardinality of the subtree where i is rooted. The z variable is used as a random index to select a reading set from replies; it is sampled from a probability-mass-function whose domain is $[1, 2, ..., length(replies)]$ and the range is $[cardinalities(1), ...]$ (note the use of the *randPMF*() external function).

The ARC_{light} version instead computes the concatenation of the readings received by children nodes plus those sensed by itself.

makeInput(*replies*, *cardinalities*)

EFFECT:
Cardinality := 0;
call *sense*();
add *LocReadings* **to** *replies*;
add 1 to *cardinalities*);
foreach $z \mid 1 \leq z \leq length(proposals)$ **do**
  $Cardinality := Cardinality + cardinalities(z)$;
$z := randPMF(length(proposals), cardinalities)$;
$RdSet := replies(z)$;

makeInput(*replies*, *cardinalities*)

EFFECT:
call *sense*();
add *LocReadings* **to** *replies*;
foreach $z \mid 1 \leq z \leq length(replies)$ **do**
  $RdSet := RdSet||replies(z)$;

Finally, tasks are the set of operations driving the execution of the automaton until it terminates. The list of those one for the $\mathcal{A}_{ARC}$ automaton is as follows:

Tasks

{tick()}
foreach $j \mid 1 \leq j \leq length(S)$ **do**
  $\{decide(RdSet, S(j).requester)_i\}$
foreach $j \mid j \in nbrs_i$ **do**
  $\{send(m)_{i,j}\}$

Table 7: The signature for the $\mathcal{A}_{ARC}$ automaton.

Type	Interface	Description
Input	$init(v)_i$	starts the protocol for process i
	$receive(m)_{j,i}$	indicates i receives a message from j
Output	$send(m)_{i,j}$	indicates i sends a message to j
	$decide(v)_i$	terminates the protocol for process i
Internal	$tick()_i$	decreases timer and increases clock
	$makeInput(v)_i$	creates the reading set to be sent to the RNG module
	$makeRequest()_i$	creates a request message
	$makeReply(r)_i$	creates a reply message for requester r
External	$encrypt(m,k)$	encrypts a message m using key k
	$decrypt(m,k)$	decrypts a message m using key k
	$sense()$	generates fresh readings using the sensing module
	$radioSend(m,dest)$	transmits a message to the specified destination using the radio module
	$estimateRTT(TTL,Delay)$	estimates the RTT given the TTL and a maximum allowed single-hop delay
	$randPMF(domain,range)$	random sampling from probability mass function (for *ARC* only)

Table 8: Description of the $\mathcal{A}_{ARC}$ automaton state variables.

Variable	Description	Accessed by
Clock	a local clock	*init*, *tick*
RdSet	the readings to be sent to RNG module if i is a requester, or to i's parent otherwise	*decide*, *makeInput*, *makeReply*
LocReadings	the reading set sensed by the i's sensor board	*makeInput*, *sense*
ToEncrypt	a message to be encrypted before transmission	*decide*, *init*, *makeReply*, *makeRequest*, *receive*
ToTransmit	a message to be immediately transmitted	*send*, *encrypt*
Secret	the array of the secret pairwise shared keys used to encrypt communications	*makeReply*, *makeRequest*, *receive*, *send*
$Delay_{max}$	the maximum allowed single-hop delay	*estimateRTT*
S	an array of sessions allowing to manage different requesters at the same time	*decide*, *init*, *receive*, *tick*
RTT	an estimation of the round trip time (fixed for *ARC* and variable for ARC_{light})	*estimateRTT*, *init*, *receive*
Nonce	a random value used to ensure authentication	*init*, *makeRequest*, *makeReply*
Cardinality	the number of nodes belonging to the tree rooted at i (useful for *ARC* only)	*makeInput*, *makeReply*
Joined	a boolean variable (*ARC* only)	*receive*
Desired	the number of readings that the requester wishes to receive from each neighbor (ARC_{light} only)	*makeReply*