TinyRNG: A Cryptographic Random Number Generator for Wireless Sensors Network Nodes

Aurélien Francillon, Claude Castelluccia

INRIA, Planète team
655, avenue de 1'Europe, 38330 Montbonnot, France
{aurelien.francillon,claude.castelluccia}@inrialpes.fr

***Abstract*— Wireless Sensors Network (WSN) security is a major concern and many new protocols are being designed. Most of these protocols rely on cryptography, and therefore, require a Cryptographic Pseudo-Random Number Generator (CPRNG). However, designing an efficient and secure CPRNG for wireless sensor networks is not trivial since most of the common source of randomness used by standard CPRNGs are not present on a wireless sensor node. We present TinyRNG, a CPRNG for wireless sensor nodes. Our generator uses the received bit errors as one of the sources of randomness. We show that transmission bit errors on a wireless sensor network are a very good source of randomness. We demonstrate that these errors are randomly distributed and uncorrelated from one sensor to another. Furthermore, we show that these errors are difficult to observe and manipulate by an attacker.**

I. TinyRNG Overview

A. Motivations

Wireless Sensors Networks (WSN) are networked constrained devices using radio communication and providing sensing services such as surveillance of a restricted area or sensing of environment. They are envisioned to be used for critical applications and/or in hostile environments (military applications, security control or natural risks prevention ...) where WSN security is a major concern. Many new security protocols are being designed. Most of them rely on cryptography, therefore, often require a good random number generator. Random generators are, for example, used to generate secret keys. Ideally, secret keys required in cryptographic algorithms and protocols should be generated with a true random number generator. However, the generation of random numbers is an inefficient procedure in most practical environments. In such situations, the problem can be ameliorated by substituting a random number generator with a pseudorandom number generator. A pseudorandom number generator (PRNG) is a deterministic algorithm which, given a truly random binary sequence of length k, outputs a binary sequence of length $l >> k$ with "appears" to be random. The input to the PRNG is called the seed, while the output generator is called a pseudorandom bit sequence. Additionally, for cryptographic applications, the generator must not be subject to observation or manipulation by an adversary. Random numbers generators based on natural sources of randomness are subject to influence by external factors, and also to malfunction.

There are typically two types of generators: hardware-based and software-based. The hardware-based generators exploit the randomness which occurs in some physical phenomena. They usually require some additional hardware and are therefore, excluded in wireless sensor networks. The software-based generators may be based on processes such as system clock, elapsed time between keystrokes, mouse movement, user input or operating system values such as system load and network statistics [7]. A well-designed software random generator should utilize as many good sources of randomness as are available, since some of them can be easily observed or manipulated. Each source should be sampled, and the sampled sequences should be combined using a randomness extractor, often a cryptographic hash function. The purpose of the extractor is to distill the true random bits from the sampled sequences. Unfortunately, most of the usual sources of entropy do not exist on a sensor (a sensor does not have a mouse, keyboard, user interface and so on). Furthermore, network timing can easily be monitored on wireless channels and the sensor clock being very slow, the timings are easy to predict. Values from the sensors of the node may be used, however, they are not always present and may provide insufficient level of randomness or low secrecy.

B. Contribution of our work

The contributions of this paper are twofold:

1) We first show that transmission bit errors on wireless sensor network are a very good source of randomness. In fact, we demonstrate that these errors are randomly distributed and uncorrelated from one sensor to another. Furthermore, we show that these errors are difficult to observe and manipulate by an attacker.
2) We design and implement a practical random generator, *TinyRNG*, for sensor networks, that uses transmission bit errors as one source of randomness.

C. System Overview

We designed and implemented a Cryptographic Pseudo-Random Number Generator that uses the received bit errors as the main source of entropy. Since bit errors are unpredictable and difficult to manipulate, we argue that they are a good source of randomness. The design of our generator was

inspired from the Fortuna system [10], but is tailored to the specific characteristics of WSN nodes.

The erroneous bytes received by a node are added into a cryptographic entropy accumulator[1]. This accumulator is built from a CBC-MAC function (see Fig. 1), which are recognized as good randomness extractors [4]. The CBC-MAC function is implemented in order to minimize the memory requirement of our system by using the same block cipher as the CPRNG. When sufficient entropy is accumulated, the accumulator is used to reseed the key of the Cryptographic Pseudo-Random Number Generator[2] (CPRNG).

The CPRNG is a block cipher (in counter mode) that encrypts a counter using the key provided at programming time and updated by re-seeding with the value generated by the accumulator. The output of this CPRNG (i.e. the encrypted counter) is then available to applications through the standard TinyOS Random interface.

II. Security Analysis

A. Attacker Model

Due to the fact that, unlike traditional workstations or server systems, WSN nodes are deeply embedded systems they do not provide memory protection and separation of rights. An attacker who is able to run code on a sensor node can have access to all its memory and therefore, to the random number generator's internal state. Consequently, it is impossible to protect the pseudo-random generator of a node that is currently under full control of the attacker.

We therefore consider the following two types of attackers:

- *Remote attackers:* The attacker did not compromise the node. Its goal is to predict or manipulate the random numbers generated by the node by external means (by observing or manipulating the communication channel for example).
- *Invasive attackers:* The attacker have compromised the node during a limited period of time, let's say from T_1 to T_2. It is able to read the memory of the compromised node during this period but cannot alter it (by adding new code for example). This can be enforced by using program integrity verification techniques [12]. The goal of the attacker is to predict the pseudo-random values that were generated either before T_1 (forward security), or after T_2 (backward security). If the node uses its pseudo-random generator to generate secret keys to encrypt its communication, the attacker should not be able to decrypt the packets that were encrypted before T_1 or after T_2.

B. Analysis of the possible Attacks

1) Remote attackers:

a) Radio eavesdropping: The attacker may try to passively eavesdrop the signal received by the victim. However, a remote attacker won't be able to gain accurate information from a remote position. If the attacker has a directional antenna, it may point it to the source of the signal. This could provide him the actual shape of the signal (such as the Error Vector Magnitude [8] of the emitted signal). However, it won't be able to get an accurate estimate of the received errors. There exist models that predict the strength of the signal when the position of the source and of the obstacles are known. However, there is no model that can accurately predict the errors that will be received by a receiver. Furthermore, even if such models would exist, they wouldn't provide accurate results, since modeling/estimating the various noise and signal (possibly coming from reflection, diffraction, other wireless networks ...) collected by the receiver is virtually impossible. Such analysis seem unlikely.

Furthermore, as shown in Section III-A, even if the attacker is very close to its target, it won't be able to observe all errors received by its victim. In fact, some errors are actually generated by the networking hardware and are not predictable.

b) Packet injection: Since it is difficult to observe and predict the pseudo-random values generated by its victim, the attacker could try to manipulate the victim's source of randomness by injecting packets. Its strategy would then be to send erroneous packets (i.e. packets with a wrong checksum). As a result, the victim would use these packets to generate its random seed. The attacker could then control the pseudo-random values generated by the victim, assuming he had knowledge of the previous internal state of the generator.

However, our experiments (Section III-B) demonstrate that if the power of reception RSSI (Received Signal Strength Indicator) or quality of reception Link Quality Indicator (LQI) [3] [3] are under a certain threshold, unpredictable bit errors will always happen. As a result, under these conditions, even if the attacker sends fake erroneous packets, the receiver will receive them with additional errors and the generated random values cannot be predicted.

2) Invasive attacks: Here we assume that the attacker has compromised the node for a limited period of time, let's say from T_1 to T_2. It was able read the memory of the compromised node during this period, but not modify it.

Note that since, as shown in the next section, bit errors are unpredictable and cannot be easily manipulated, the attacker does not have any information about the bit errors that happen before T_1 and after T_2. As a result, it won't be able to predict the pseudo-random values, and therefore, the secret keys generated by the victim before $T_1 - \delta_1$ and after $T_2 + \delta_2$, where δ_1 and δ_2 are dependent on the reseed period of the generator.

[1] As explained later, we actually use two accumulators

[2] During a reseed the previous key is added to the accumulator and the output of the accumulator is then used as the new key.

[3] RSSI is a value provided by the CC2420 [3] wireless network device which is an indicator of signal reception level. The LQI gives an indication of the quality of signal reception, and is computed from the average symbol correlation of the whole packet data, excluding the preamble and start of frame delimiter. This gives a fairly precise indication of the capability of reception of the radio device.

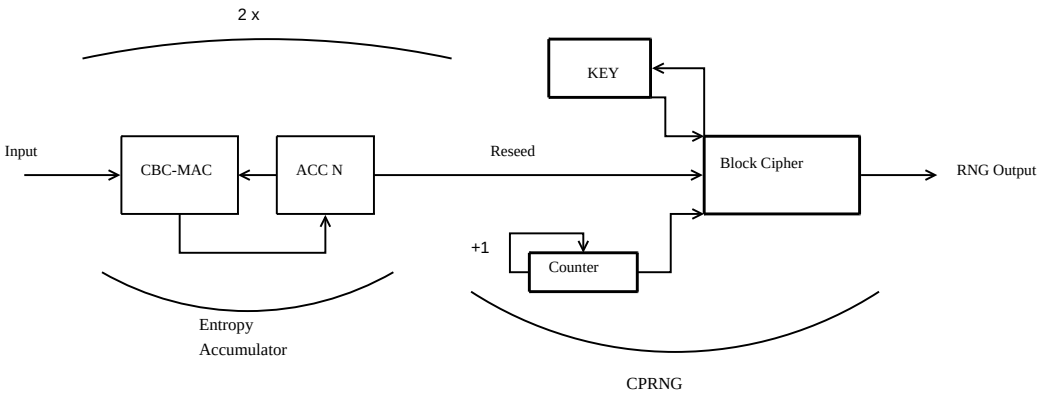


Fig. 1. TinyRNG Block diagram

III. Analysis of bit errors on wireless channels

The goal of this section is to experimentally show that the bit errors received by a node are difficult to observe/predict and to manipulate.

A. Errors cannot be predicted

In order to achieve this goal, we consider a very powerful attacker. The attacker has physical access to the antenna of the victim node and is trying to predict the errors that are received by the victim.

1) Experimental setup: Our experimental setup is the following: We connect two motes (the victim mote and the attacker mote) to the same antenna. This is performed using a 2-port power divider from Pulsar [13] as illustrated by Fig. 2. The divider attenuates the received signal by 2db. A third mote then sends 10^6 packets at a rate of 4 packets per second. All data received by the two motes are logged on a computer. Note that, in this experiment, all the packets sent by the third node are identical and known to the receivers. Erroneous bits are detected by comparing the received data to the expected one.

2) Data Analysis: In order to analyse the collected data, we generated two strings of bytes. The first one, F, contains all the bytes that were received erroneously by the victim mote. The second string, G, contains all the bytes that were received erroneously by the attacker. We then computed the auto-correlation and cross-correlation of the F and G strings. The correlation, at lag i is defined as:

$$RawCorrelation(f,g)_i = \sum_j f_j * g_{i+j}$$

This value is then normalized such that the correlation of two identical sequences at $lag = 0$ is equal to 1. [4]

Figure 3(a) displays the auto-correlation of the the errors collected respectively by the victim and the attacker. These results show that the errors received by each node are uncorrelated.

[4]To compute the correlation values we used the Matlab "xcorr" function with option "coeff" for normalization.

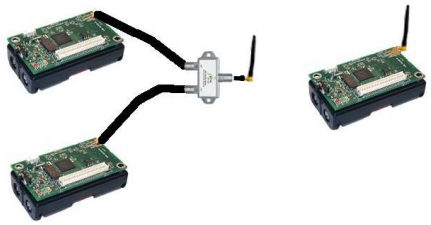


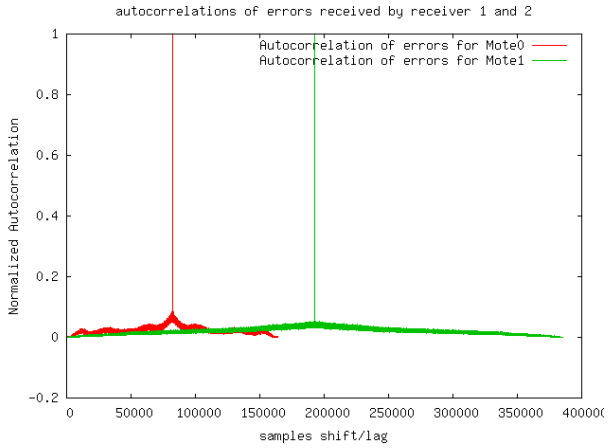
Fig. 2. Experimental Setup

As a result, if the attacker has access to the received bit errors received by the victim during a period of time, it won't be able to predict the bit errors that will appear next. This is a very important result, since it shows that both forward and backward security can be provided.

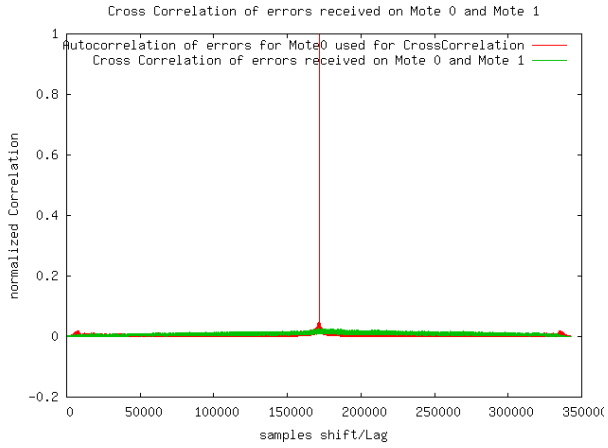
Figure 3(b) displays the cross-correlations of erroneous data received by the victim and the attacker. It shows that the errors received by the attacker mote are completely uncorrelated from the errors received by the victim mote. The motes are often getting errors in different positions. i.e. for 4.4% of the samples, the victim mote gets an erroneous packet while the attacker received the packet correctly. Furthermore, in 6.2% of the samples, the victim received the packet correctly while the attacker didn't receive anything. This makes 10.6% of packet errors that are unique to the victim and that cannot be observed by the attacker. We believe that many of these errors are actually due to the network device. These results give evidence that it is very difficult for an attacker to actually predict the receiving errors of its victim. This is an important result, since it shows that bit errors are a good source of *secret* randomness.

B. Bit errors cannot be manipulated

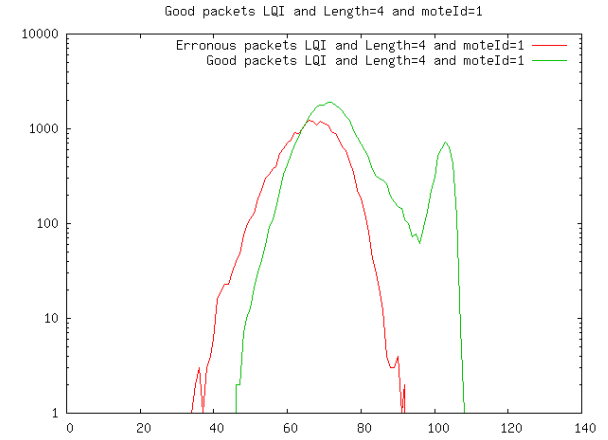
The goal of this section is to give evidence that bit errors cannot easily be manipulated. In order to demonstrate this result, we measured and plotted the LQI of the packets that are



(a) Auto-correlation



(b) Cross-correlation



(c) LQI measurement

Fig. 3. *Analysis of Errors*

received with errors and of packets that are received without errors. The results are displayed on Fig. 3(c). They show that packets that are received with an LQI smaller than 45 are **always** erroneous. Note that our experimentations provide the same result with RSSI: packets received with a raw RSSI smaller than -50db are also **always** erroneous. These results show that an attacker cannot control all the errors that are on the channel. If these packets' LQI is smaller than 45, the generated errors cannot be controlled, and therefore observed, by the attacker. TinyRNG makes use of this property. It uses two accumulators. The first one uses the received packets with wrong CRC. The second one uses the received packets with error and with a LQI smaller than 45.[5] If the attacker sends malicious packets with wrong CRC at a relatively high signal level, they will be added to the first accumulator. However, if the attacker tries to corrupt the second accumulator with malicious packets, he will have to reduce the power of emission under a level at which additional transmission errors will occur in the packets. The first accumulator will be filled up more quickly and will therefore be used for reseed more frequently. The second one is slower but provides a much better security against active attacks.

IV. Implementation

A. Internal state initialization

The initial state of the random number generator is given at programming time by an initial key for the block cipher. This key is generated on the development computer, which is assumed to have a good random number generator. This may lead to problems with large scale code distribution systems such as deluge [5]: all nodes will be remotely programmed with the same initial state. Therefore, before deployment of such a large scale network, a seeding phase is performed with a dedicated program image. This image contains a list of seeds $S_0, S_1, ..., S_i$ for node 0 to i. When the program executes on node i the seed S_i is stored in the internal EEPROM [1] of node i. Note that if the nodes don't share an apriori secret with the programmer, this step has to be performed in a trusted environment. On the other hand, if nodes are sharing a key with the programmer, the image can include per-node encrypted seeds. This image is eventually overwritten by the actual program, removing all traces of the initial seeds other than the values in the EEPROM. When a node is booting, it uses its unique node identifier as first reseed, then it saves a few bytes of random numbers in the EEPROM, overwriting the previous seed. This ensures that nodes won't be vulnerable to *reboot attacks*, because the next reboot will use the new seed value. In order to reflect the evolution of the entropy of the internal state after a certain amount of time, a new seed should be saved before shutdown of a node. The seed may also be saved on a periodic basis to protect against unclean shutdown such as watchdog timeouts or malicious reboots. Furthermore, it is computationally impossible to retrieve the previous initial

[5]This value may need to be keep dynamically in order to adapt to different radio devices. Indeed, similar devices can have different limit of reception.

Configuration	ROM memory usage (Bytes)	RAM memory usage (Bytes)
MICA2, RandomLFSR	7686	302
MICA2, TinyRNG	11086	471
Overhead	3400 (44%)	169 (56%)
MICA2, TinySec, RandomLFSR	16382	559
MICA2, TinySec, TinyRNG	17774	732
Overhead	1392 (8.5%)	161 (28%)
MICAz, RandomLFSR	6132	244
MICAz, TinyRNG	9832	416
Overhead	3700 (60%)	172 (70%)

Fig. 4. Main memory consumption Comparison TinyRNG vs TinySec

Component	RAM memory usage (Bytes)
Accumulators States	2*32
Cipher Context (CTR mode)	20
Precomputed random numbers (minimal size)	8
Counter	8
Keys	2*10
Total	120

Fig. 5. Main memory consumption sources in TinyRNG

state of the random number generator of the nodes, as this would be equivalent to breaking the underlying block cipher.

B. Memory usage

TinyRNG has been designed to minimize its memory usage. In this section, we compare TinyRNG and RandomLFSR memory footprints, i.e. the number of ROM and RAM bytes used by each of these programs. RandomLFSR is a simple random number generator for TinyOS. Note that RandomLFSR is not cryptographic and does not provide secure random numbers, the randomness of its outputs have also been discussed. We also compare memory usage of TinyRNG with TinySec [9] for MICA2 motes.[6].

To evaluate the memory footprint of TinyRNG we used a program which periodically uses random numbers. Figure III-B shows the memory usages for various configurations and outlines the overhead of TinyRNG. It shows that TinyRNG ROM and RAM memory costs are respectively 44% and 56% higher than RandomLFSR memory footprints. However, these cost overheads reduce to 8.5% and 28% respectively, if TinySec is also present in the mote, because TinyRNG reuses many of its modules.

C. Energy consumption

Energy consumption is a major concern for WSN nodes. Figure IV-C details the main sources of energy consumption in TinyRNG and their timings. The major operations are the initialization, entropy accumulation, reseed and random numbers generation phase.

[6]Unfortunately TinySec is not available for MICAz motes on which we developed the TinyRNG module, our implementation on MICA2 motes is in an early state, however, final results should not be significantly different than the ones presented here

1) Initialization: When the node is booting, it reads the previously stored seed from EEPROM, performs a reseed and generates a new seed from the random number generator. This new seed is then written back to the internal EEPROM memory. This operation performs write access to the Atmega128 internal EEPROM [1], which comes at a higher energy consumption cost than any other operation in TinyRNG by two orders of magnitude ($4.85mJ$). However, this is an infrequent operation as its performed only at boot time. This step may also be performed by a timer at a very low frequency in order to update the saved seed, such that further reboots are taking into account the evolution of the internal state. The energy cost of this operation should be taken into account when selecting the timer period. The energy consumption of the accumulators and PRNG initialization is negligible with respect to that of seed write. Another fact is that EEPROM write may be impossible if battery power is too low, which may lead to successful *reboot attacks*. If the mote boots under those conditions TinyRNG will be functioning in a degraded mode until properly reseeded.

2) Entropy Accumulation: During the entropy accumulation phase, each erroneous received packet is used to update the accumulator state. This is done by adding the packet to the incremental CBC-MAC state, which itself uses Skipjack. The energy consumed by this step is similar to the energy that TinySec [9] would spend to check the MAC of a correctly received packet.

3) Reseed: For the reseed phase, the current key of the CPRNG block cipher is fed back into the accumulator, and the output of the accumulator is used as the new key. The accumulators are eventually reinitialized. The energy consumption of this step is dominated by the energy consumption of the block cipher encryption. Since this operation is not frequent,

Configuration	Process time		Energy Consumption	
Random numbers generation				
RandomLFSR request 64 random bits	28.4	μs	0.75	μJ
TinyRNG request 64 random bits	440	μs	11.4	μJ
Overhead (ratio)	15.5		15.2	
Initialization				
TinyRNG Initialization of CPRNG (with r/w EEPROM operations)	144	ms	4.85	mJ
TinyRNG Initialization of CPRNG (without EEPROM operations)	2.15	ms	53.7	μJ
Accumulators Initialization	1.47	μs	36.3	μJ
Periodic events				
Reseed	1.13	ms	28	μJ
TinyRNG entropy Accumulation	2.16	ms	56.3	μJ

Fig. 6. Energy consumption Comparison TinyRNG vs RandomLFSR on MICAz platform

its impact on the overall consumption is limited.

4) Random number generation: Generation of random numbers is performed by using the block cipher in counter mode. One block operation is required for each 64 random bits generation. This means that the energy consumption is proportional to the number of consumed random bits. Because this step is about 15.5 times slower than with a simple random number generator such as TinyOS's LFSR, it should be used with care when timing sensitive operations such as interrupts handlers or network drivers. However, some precalculation of random numbers may be possible if some memory pool is available for buffering. This would ensure fast access to good quality random numbers. If ever the amount of random numbers requested is too high and it would be energy prohibitive to use TinyRNG, or the precomputed random numbers pool is empty, fast access to random numbers can be provided by the LFSR initialized with a seed from the output of TinyRNG. However, this should not be used for keying material.

V. Related Work

Gutmann [7] give advice on how to properly mix the entropy collected in entropy pools and implementation advice for random number generators in software with a system point of view. In [10] (extended in [6]) Kelsey, Ferguson and Schneier give some advice on how to implement a random number generator resistant to state compromise attacks and incremental attacks. Our design is derived from this. [2] presents a formal construction for random numbers generators and formally defines forward and backward security. Barak et al. advocates for periodic reseed, with low period, or a very conservative entropy estimate rather than trying to estimate entropy gathered from the point of view of the attacker, which is unfeasable by a computer program. The construction proposed is similar to that of [10]. Mutaf suggests [11] to use timing of radio link noise level variations. This approach is not suitable for WSN nodes since there is no way to constantly get the accurate timing of the noise level changes efficiently in terms of power consumption (i.e. it needs constant sampling of the noise level as well as accurate timestamp). In [15] Seznec et al. propose to gather entropy from variation in execution time of algorithms due to the micro states in the central processing unit. However, this won't produce much entropy on sensor networks due to the low complexity of the processors and the very limited processing activity on such devices, as well as the simplicity of the threading on these platforms. Finally, the random number generators proposed for wireless sensor networks nodes like [14] or included in TinyOS are not suitable for cryptographic usage.

VI. Conclusion

We have presented TinyRNG, a new cryptographic pseudo-random generator that is well adapted to wireless sensors. An implementation have been made and evaluated, and will be made publicly available. Our generator uses the bit errors of the received packets as a source of randomness. We show that bit errors are unpredictable and cannot be easily modified.

Furthermore, we believe that because WSN networks are often composed of cheap devices and are deployed in open and harsh environments, bit errors, due either to the environment or the reception hardware, are frequent. Even in controlled and closed environments, bit errors happen in wireless transmissions. This was illustrated by our in-lab experiments. Furthermore, since transmitting packets is the most energy consuming operation and since WSNs are dense, each node tends to minimize their transmission power and, therefore, increases the number of erroneous bits.

VII. Acknowledgments

We would like to thank Gérard Baille, Roger Pissard-Gibollet and Jean-François Cuniberto from the SED team at INRIA as well as Maté Soòs for their kind help.

The work described in this paper is based on results of IST FP6 STREP UbiSec&Sens (http://www.ist-ubisecsens.org). UbiSec&Sens receives research funding from the European Community's Sixth Framework Programme. Apart from this, the European Commission has no responsibility for the content of this paper. The information in this document is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

REFERENCES

[1] Atmel. Atmega128(l) datasheet, doc2467: 8-bit microcontroller with 128k bytes in-system programmable flash.

[2] B. Barak and S. Halevi. A model and architecture for pseudo-random generation with applications to /dev/random. Cryptology ePrint Archive, Report 2005/029, 2005.

[3] Chipcon. Cc2420, 2.4ghz ieee 802.15.4 / zigbee-ready rf transceiver.

[4] Y. Dodis, R. Gennaro, J. Håstad, H. Krawczyk, and T. Rabin. Randomness extraction and key derivation using the CBC, Cascade and HMAC modes. LNCS 3152, 2004.

[5] P. K. Dutta, J. W. Hui, D. C. Chu, and D. E. Culler. Securing the deluge network programming system. In *IPSN*, pages 326–333, 2006.

[6] N. Ferguson and B. Schneier. *Practical Cryptography*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[7] P. Gutmann. Software generation of practically strong random numbers. In *7th USENIX Security Symposium, San Antonio, Texas*.

[8] IEEE. P802.15.4/d18 draft standard: Low rate wireless personal area networks.

[9] C. Karlof, N. Sastry, and D. Wagner. Tinysec: A link layer security architecture for wireless sensor networks. In *SensSys, ACM Conference on Embedded Networked Sensor Systems*, 2004.

[10] J. Kelsey, B. Schneier, and N. Ferguson. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator. In *Selected Areas in Cryptography*, number Generators, pages 13–33, 1999.

[11] P. Mutaf. *True random numbers from Wi-Fi background noise*. http://www.freewebs.com/pmutaf/iwrandom.html.

[12] T. Park and K. G. Shin. Soft tamper-proofing via program integrity verification in wireless sensor networks. *IEEE Transactions on Mobile Computing*, 2005.

[13] Pulsar Microwave Corporation. Pulsar power divider p2-20-414.

[14] D. Seetharam and S. Rhee. An effcient random number generator for low-power sensor networks.

[15] A. Seznec and N. Sendrier. Havege: A user-level software heuristic for generating empirically strong random numbers. *ACM Trans. Model. Comput. Simul.*, 13(4):334–346, 2003.