Security Flaws in an Efficient Pseudo-Random Number Generator for Low-Power Environments

Pedro Peris-Lopez[1], Julio C. Hernandez-Castro[2], Juan M.E. Tapiador[3], Enrique San Millán[4], and Jan C.A. van der Lubbe[1]

[1] Department of Information and Communication, Delft University of Technology, The Netherlands
[2] School of Computing, Buckingham Building, Lion Terrace, Portsmouth PO1 3HE, United Kingdom
[3] Department of Computer Science, University of York, Heslington, York, YO10 5DD, United Kingdom
[4] Department of Electrical Engineering, University Carlos III of Madrid, 28911 Leganés, Spain

p.perislopez@tudelft.nl, julio.hernandez-castro@port.ac.uk, jet@cs.york.ac.uk, quique@ing.uc3m.es, j.c.a.vanderlubbe@tudelft.nl

Abstract. In 2004, Settharam and Rhee tackled the design of a lightweight Pseudo-Random Number Generator (PRNG) suitable for low-power environments (e.g. sensor networks, low-cost RFID tags). First, they explicitly fixed a set of requirements for this primitive. Then, they proposed a PRNG conforming to these requirements and using a free-running timer [9]. We analyze this primitive discovering important security faults. The proposed algorithm fails to pass even relatively non-stringent batteries of randomness such as ENT (i.e. a pseudorandom number sequence test program). We prove that their recommended PRNG has a very short period due to the flawed design of its core. The internal state can be easily revealed, compromising its backward and forward security. Additionally, the rekeying algorithm is defectively designed mainly related to the unpractical value proposed for this purpose.

Keywords: Sensor networks, RFID, PRNG, security, cryptanalysis.

1 Settharam and Rhee PRNG

In 2003, Rhee *et al.* designed an ultra-low power sensor networking platform, named i-Bean Network [7]. In this platform, sensors must support a Pseudo-Random Number Generator (PRNG) on-board for various purposes such as random transmissions delays or the generation of random packet sequence numbers. The use of Linear Congruential Generators (LCGs) or Lineal Feedback Shift Registers (LFSRs) could be appropriate due to their low hardware requirements (circuitry, memory and power consumption), but these generators are completely insecure. An alternative can be the use of a Cryptographically Secure Pseudorandom Number Generators (CSPRNG), but they are very exigent in terms of hardware demands, being unpractical in this kind of environments.

Finally, standard cryptographic primitives such as a block cipher with a secret key working in counter mode or a hash function with a secret key operating in output feedback mode, can be used to build a PRNG. The problem here is again that these kind of constructions also exceed by far the capabilities of constrained devices. Motivated by this necessity, Seetharam and Rhee proposed an *ad hoc* simple PRNG based on a free-running timer [9]. This PRNG not only can be used for the i-Bean network, but it is suitable for other low-power environments. Prior to their design of the PRNG, the authors fixed a set of requirements that should be satisfied by their proposed generator, or any other design, for being useful within the i-Bean network environment:

- The generator must be efficient. Efficiency is defined in terms of resources, temporary requirements and memory. This property is translated into the following requirements: 1) It does not require any multiplication or division operations. It would be desirable if the generation could be achieved using just the logical operations; 2) The number of steps required for generating a single random number must be less than 10; 3) The amount of code memory used must be less than 50 bytes.
- The generator must produce a uniform distribution of random bytes (numbers in the interval [0,255]).
- The generator must not use specific features of any microcontroller, in this way being easily portable to other hardware platforms.

Upon establishing the above requirements, the authors proposed an 8-bit PRNG based on a free running timer (SR-PRNG in short). To obtain a fresh random number an XOR between the current value of the timer and the key is computed ($rv = tv \oplus key$). Next, the key and the timer values are updated. Specifically, one's complement of the timer becomes the new key ($key = \sim tv$) and the one's complement of the new random number becomes the next time value ($tv = \sim rv$). Additionally, the key is updated after the generation of K consecutive random numbers. To accomplish this task, the checksum of the received/transmited packets is used. The C-code of the proposed PRNG is included below. Exclusive-OR operation and one's complement are symbolized by "ˆ" and "∼" respectively.

```
/* Initialize Key to the ID of the node */
unsigned char key = nodeID;

/* Seetharam et al.'s PRNG */

unsigned char SR-PRNG()
{  unsigned char rv = 0;
   unsigned char tv = 0;

   tv  = get_timer();

   rv  = tv ^ key;
   key = ~tv;
```

```
    tv  = ~rv;

    set_timer(tv);

    return rv;
}
```

2 Statistical Analysis of SR-PRNG

To get any evidence of the security of a PRNG, it should be subjected to a variety of statistical tests designed to detect the specific characteristics expected from random sequences. In fact, there are different battery of tests for the evaluation of randomness. In 1995, Marsaglia introduced the Diehard tests which are a battery of stringent statistical tests for measuring the quality of a set of random numbers [5]. ENT is also another test battery, which includes the chi-square test [10]. A dedicated suite of randomness tests suitable for the evaluation of PRNG used in cryptography was proposed by the National Institute of Standards and Technology (NIST) in 2001 [8]. Recently, Sexton proposed a new series of tests [1], whose interpretation is similar to that of Diehard. Passing these battery tests is a necessary but not a sufficient condition for a generator to be secure. On the other hand, systematically passing the NIST and Diehard batteries provides strong evidence in favor of a good degree of output randomness.

In [9], authors used ENT for the evaluation of their PRNG. Specifically, ENT performs a variety of tests to the stream of bytes stored in a file. The entropy, chi-square test, arithmetic mean, Monte Carlo value for Pi, and serial correlation are the values computed. We have implemented the proposed PRNG in order to analyze its output.

There is a relevant point that is unclear in the original paper. Authors specified that the key is updated with the 8-bit Cyclic Redundancy Check (CRC) values of the transmitted and received packets. However, they do not describe any property of the messages transmitted and received in the network. In other words, we do not know the exact entropy introduced by this source.

Additionally, authors neither describe the rekeying algorithm in the paper nor facilitate us this information by personal communication. We do not know if, for example, the new key is simply the CRC of the messages passed to the channel, or instead the new key is the result of computing an XOR between the old key and the CRC.

Finally, the authors do not specify how frequently is necessary to apply the rekeying. This is a crucial point, as we demonstrate in Section 3.1. For evaluation purposes, the authors fixed this value to 10, but no justification for this number appears on the paper. In order to be able to analyze the output provided by this PRNG, we first describe the rekeying algorithm used in our experimentation (Python code is available in Appendix A).

Rekeying algorithm: In the initialization phase, we initialize a string variable A to a random value. Once the generator computes K random values ($rk = K$), the key is updated. We randomly changed the 3 most significant bits of the

Table 1. ENT Statistical Tests

Test	$rk = 10$	$rk = 100$
Entropy	7.707212	7.310092
Chi-square Test	477179.13 (0.01%)	477179.13 (0.01%)
Arithmetic Mean	126.1655	113.1528
Monte Carlo Value for Pi	2.940398806 (6.40%)	3.094900134 (1.49%)
Serial Correlation Test	-0.026964	-0.150160

variable A and compute its CRC. The key is finally updated by computing the XOR between the actual key and the result obtained from the CRC.

Once the rekeying algorithm is defined, we generate two files of 300MBytes, in the first of these files the rekeying is fixed to 10 ($rk = 10$) and in the second to 100 ($rk = 100$). If $rk > 100$, the results are catastrophic and are laking in interest. We analyze each of these files with the ENT battery as Seetharam and Rhee did. The results obtained are presented in Table 1.

First we focus on the results obtained when the rekeying factor is fixed to 10. There are many output values that offer a strong evidence of the non-randomness of the analyzed output:

– The arithmetic mean should be around 127.5 if the data were close to random. The obtained value points out that the output has a strong bias.
– The error in the Monte Carlo estimation of Pi value is huge.
– The serial correlation test evidences a slight dependence between each output byte and the previous output byte. This dependence is higher (in absolute value) as the rekeying value is incremented (see Section 3.1).
– The chi-square test is specially revealing: the percentage should be interpreted as the likelihood of the tested sequence coming from a uniform distribution. As the percentage obtained is inferior to 1% (around 1 in 10000), we can safely conclude that the studied sequence is almost certainly not random. Additionally, the measured chi-square statistic is astronomically larger than that expected (477179.1 against 255.3).

The results obtained with the rekeying factor fixed to 100 are indisputable:

– The density of information (entropy) is further reduced in comparison with the above case ($rk = 10$). Additionally, the value obtained is significantly far from the optimal value (8.0).
– The arithmetic mean indicates that the output has a very strong bias. In other words, the probabilities of ones and zeros are significantly different.
– The high value of the serial correlation coefficient points out the absolute necessity of a low rekeying factor. This factor would have to be fixed to an extremely low value, as we will see in Section 3.1. This fact evince that the core of the PRNG proposed by Seetharam and Rhee was not properly designed. Additionally, the highly negative correlation shows that the computation of complements in the PRNG is still easily observable in its output.

We have also analyzed the above two files with the Diehard battery. We will omit the results here due to their limited interest and just mention the main conclusions. In summary, the generator dramatically fails to pass each of the tests included in Diehard, even when the rekeying factor is fixed to an unpractically low value ($rk = 10$). The two sequences ($rk = 10$ or $rk = 100$) do not pass a single test, obtaining 0.0 or 1.0 for all the test p-values, and an overall p-value of 0. From all of the above, we can safely conclude that the analyzed sequences (and their generators) consistently failed the Diehard battery of tests at the 0.05 significance level.

3 Cryptanalysis of SR-PRNG

In this section we present the cryptanalysis of the PRNG proposed by Seetharam and Rhee. First, we show that their PRNG has an extraordinarily short period. Then, we demonstrate how the internal state of the generator can be easily disclosed. The above mentioned properties are extremely bad for, respectively, randomness and security reasons.

3.1 Period Evaluation

We show in the following that the period of this generator is extremely short. Specifically, every three executions the same value is generated again. In other words, if the PRNG is in the state $(tv[n], k[n])$, the following sequence is generated: $\{rv[n+1], rv[n+2], rv[n+3]\}$, $\{rv[n+1], rv[n+2], rv[n+3]\}$, etc. This can be generalized as: $rv[n+r] = rv[n + (r \mod 3)]$ for any r.

Theorem 1. *SR-PRNG returns to the same internal value each 3 iterations:*

$$\left.\begin{array}{r} key[n] = key[n+3*m] \\ tv[n] = tv[n+3*m] \end{array}\right\} For\ any\ integer\ m \tag{1}$$

Proof. We start observing the output and the internal state of three consecutive executions:

$$\begin{aligned} &Iteration \quad n \\ rv[n] &= tv[n] \oplus key[n] \\ key[n+1] &= \sim (tv[n]) \\ tv[n+1] &= \sim (rv[n]) \end{aligned} \tag{2}$$

$$\begin{aligned} &Iteration \quad n+1 \\ rv[n+1] &= tv[n+1] \oplus key[n+1] \\ key[n+2] &= \sim (tv[n+1]) \\ tv[n+2] &= \sim (rv[n+1]) \end{aligned} \tag{3}$$

$$\begin{aligned} &\mathit{Iteration} \quad n+2 \\ &rv[n+2] = tv[n+2] \oplus key[n+2] \\ &key[n+3] = \sim (tv[n+2]) \\ &tv[n+3] = \sim (rv[n+2]) \end{aligned} \tag{4}$$

Employing the above 3 equations recursively, it easy to find that

$$\begin{aligned} key[n+3] &= \sim (tv[n+2]) \\ &= \sim (\sim (rv[n+1])) = rv[n+1] \\ &= tv[n+1] \oplus key[n+1] \\ &= \sim (rv[n]) \oplus \sim (tv[n]) \\ &= rv[n] \oplus tv[n] \\ &= key[n] \end{aligned} \tag{5}$$

and that

$$\begin{aligned} tv[n+3] &= \sim (rv[n+2]) \\ &= \sim (tv[n+2] \oplus key[n+2]) \\ &= \sim (\sim (rv[n+1]) \oplus \sim (tv[n+1])) \\ &= \sim (rv[n+1] \oplus tv[n+1]) \\ &= \sim (key[n+1]) \\ &= \sim (\sim (tv[n])) \\ &= tv[n] \end{aligned} \tag{6}$$

□

This proves that after 3 iterations the generator returns to the same internal values. Therefore, the same sequence is generated again.

The authors proposed that the key have to be changed after the generation of K random values. If the rekeying is performed each K iterations ($rk = K$), the Equation 1 can be rewritten as

$$\left.\begin{aligned} key[n] &= key[n+3*m] \\ tv[n] &= tv[n+3*m] \end{aligned}\right\} \text{If} \quad n+3*m < K \tag{7}$$

But the very short period is not mitigated by rekeying unless this is performed every 3 iterations. This value is unpractical and would overload the random number generation. The authors do not seem to be aware of this design problem, as they suggest the rekeying factor to be 10.

Rekeying is generally used in cryptography to limit the amount of data encrypted under the same key. A key exchange protocol is usually employed to negotiate the new key. In our case, the rekeying is used to update one of the internal values of the PRNG state. This process should introduce enough randomness in the new key, but the authors proposed the *CRC* of the transmitted/received packets, and as source of randomness it is not good enough. This choice presents three main problems: 1) The *CRC* is not a good source of randomness, as revealed by the statistical properties of the sequences analyzed; 2) The rekeying process increments both the computational load and the power consumption, and significantly reduces the throughput. To avoid these drawbacks, a high rekeying factor is usually used [3], exactly the opposite approach than the extremely low value suggested by the authors; 3) Initially, the key is set with the identification number of the node. In applications where this identification number might change (e.g. RFID systems), if this value is updated, the central back-end database and the reader get into a desynchronized state.

3.2 Disclosure of the Internal State

In this section we show how the internal secret state of the PRNG can be recovered under the very realistic assumption that only two consecutive outputs are eavesdropped. We demonstrate that SR-PRNG does not provide neither soft-forward nor soft-backward security.

Definition 1. ***Soft-Forward security*** *is the property that guarantees that an adversary listening the outputs provided by a PRNG (i.e.* $rv[n+1]$, $rv[n+2]$*) is not able to predict the next outputs (i.e.* $rv[n+3]$, $rv[n+4]$*, etc). In general terms,*

$$P_{Adv}(rv[n+k+m]|\{rv[n+i]\}_{i=1}^{k}) = \varepsilon \tag{8}$$

for $m = 1, 2, \ldots$ *and* ε *some negligible value and where* P_{Adv} *is the probability.*

Definition 2. ***Soft-Backward security*** *is the property that guarantees that an attacker listening the outputs provided by the PRNG (i.e.* $rv[n+1]$, $rv[n+2]$*) is not able to determinate the previous state (*$s[n+1] = (tv[n+1], key[n+1])$*) after a new state (*$sv[n+2] = (tv[n+2], key[n+2])$*) has been reached:*

$$P_{Adv}(sv[n+1]|\overline{sv[n+2]}, rv[n+1], rv[n+2]) = \varepsilon \tag{9}$$

where $\overline{sv[n+2]}$ means that $sv[n+2]$ state has been reached by the PRNG but its unknown to the attacker, and ε is a negligible value.

Theorem 2. *An adversary can recover the internat state of SR-PRNG after the eavesdropping of two consecutive outputs* $\{rv[n], rv[n+1]\}$*:*

$$\left.\begin{array}{r} tv[n] = rv[n] \oplus rv[n+1] \\ k[n] = rv[n+1] \end{array}\right\} \textit{For any integer } n \tag{10}$$

Proof. We start observing the output and the internal state of two consecutive executions.

$$\begin{aligned} &Iteration \quad n \\ rv[n] &= tv[n] \oplus key[n] \\ key[n+1] &= \sim (tv[n]) \\ tv[n+1] &= \sim (rv[n]) \end{aligned} \tag{11}$$

$$\begin{aligned} &Iteration \quad n+1 \\ rv[n+1] &= tv[n+1] \oplus key[n+1] \\ key[n+2] &= \sim (tv[n+1]) \\ tv[n+2] &= \sim (rv[n+1]) \end{aligned} \tag{12}$$

Next, we show how an attacker is able to obtain the actual state of the generator. Once the state is known, future outputs can be computed, compromising the soft-forward security. Applying the above equations,

$$\begin{aligned} key[n+2] &= \sim (tv[n+1]) \\ &= \sim (\sim rv[n]) \\ &= rv[n] \end{aligned} \tag{13}$$

$$tv[n+2] = \sim (rv[n+1]) \tag{14}$$

Finally, we demonstrate how the attacker is also able to acquire the previous state of the generator compromising the soft-backward security:

$$\begin{aligned} rv[n+1] &= tv[n+1] \oplus key[n+1] \\ &= \sim (rv[n]) \oplus \sim (tv[n]) \\ &= rv[n] \oplus tv[n] \end{aligned} \tag{15}$$

From Equation 15,

$$tv[n] = rv[n] \oplus rv[n+1] \tag{16}$$

After the previous timer value is obtained, the last key value is easily obtained applying Equation 16:

$$\begin{aligned} k[n] &= rv[n] \oplus tv[n] \\ &= rv[n] \oplus rv[n] \oplus rv[n+1] \\ &= rv[n+1] \end{aligned} \tag{17}$$

Equations 16 and 17 can be used with $n = 0$, thus revealing the secret seed:

$$k[0] = rv[1] \tag{18}$$

$$tv[0] = rv[0] \oplus rv[1] \tag{19}$$

$\square$

Barak and Halevi proposed a formal model and a simple architecture for robust pseudorandom generators [2]. In this model backward/forward means that future/past output is secure. We switched this notation to be consistent with the conventional one. Three properties are demanded to these kind of architectures:

– Resilience: the output should look random to an observer with no knowledge of the internal state. This should hold even if the observer has complete control over the data used to refresh the internal state.
– Backward security: past output of the generator should look random to an observer, even if he knows the internal state at a later time.
– Forward security: future generator output should look random, even to an observer with knowledge of the current state, provided that the generator is refreshed with data with enough entropy.

As shown in Section 2, the output provided by SR-PRNG does not look like random at all. Backward security is compromised even if the internal state of the PRNG is not revealed (soft-backward security is not guaranteed). Therefore, SR-PRNG is not a robust PRNG, which might be necessary for designing protocols with forward and backward untraceability. In fact, this it is a sufficient but not necessary condition as Phan *et al.* showed in [6].

4 Conclusions

In this paper, we present the cryptanalysis of a lightweight PRNG suitable for low-power environments (e.g. sensor networks). We discover important security faults concerning both its core function and rekeying algorithm.

To design the core function, the authors limited the set of operations supported on-chip (i.e. PRNG) to bitwise operations (i.e. addition modulo 2 or one's complement). The bad property from the point of view of security is that all of these operations are triangular functions [4]. That is, the bit in position i in the output only depends on bits $j = 0, \dots , i$ of the input words, instead of all input bits. Furthermore, the composition of triangular operations always results in a triangular function. These undesirable characteristics greatly facilitates their successful analysis.

According to the rekeying algorithm two unfortunate decisions were taken. First of all, the CRC of the transmitted packets has a very low entropy, and additionally, these packets are really easy to alter by an active attacker. Secondly, a refreshment period of 10 iterations makes not sense; it would imply rekeying each 50 milliseconds, which computationally is too demanding.

Our next logical step is the great challenge of designing a secure lightweight PRNG under the requirements stated by Settharam and Rhee. Some requirements may be redefined because its specification needs further clarification (e.g. the definition of a single step).

References

1. David Sexton's battery (2005), `http://www.geocities.com/da5id65536`
2. Barak, B., Halevi, S.: A model and architecture for pseudo-random generation with applications to /dev/random. In: ACM Conference on Computer and Communications Security, pp. 203–212 (2005)

3. Bernstein, D.J.: Salsa20 specifications (2005), http://www.ecrypt.eu.org/stream/
4. Klimov, A., Shamir, A.: Cryptographic applications of T-functions. In: Matsui, M., Zuccherato, R.J. (eds.) SAC 2003. LNCS, vol. 3006, pp. 248–261. Springer, Heidelberg (2004)
5. Marsaglia, G.: The Marsaglia Random Number CDROM Including the DIEHARD Battery of Tests of Randomness (1996), http://stat.fsu.edu/pub/diehard
6. Phan, R.C.-W., Wu, J., Ouafi, K., Stinson, D.R.: Privacy Analysis of Forward and Backward Untraceable RFID Authentication Schemes (2008), http://www.cacr.math.uwaterloo.ca/~dstinson/papers/bfrfid-2.pdf
7. Rhee, S., Seetharam, D., Liu, S., Wang, N., Xiao, J.: i-Bean Network: An Ultra-Low Power Wireless Sensor Network. In: Proceedings of UBICOMP 2003 (2003)
8. Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A., Dray, J., Vo, S.: A statistical test suite for random and pseudorandom number generators for cryptographic applications. NIST special publication 800-22 (2001), http://csrc.nist.gov/rng/
9. Seetharam, D., Rhee, S.: An Efficient Pseudo Random Number Generator for Low-Power Sensor Networks. In: Proceedings of LCN 2004, pp. 560–562. IEEE Computer Society, Los Alamitos (2004)
10. Walker, J.: Randomness Battery (1998), http://www.fourmilab.ch/random/

Appendix A

This is the source code of our implementation in Python.

```
from random import *
from scipy import *

#define the CRC
from crc_algorithms import Crc
crc = Crc(width = 8, poly = 0x7, reflect_in = False,
xor_in = 0x0, reflect_out = False, xor_out = 0x0)

# - Begin Program -

f=open('output.dat', 'wb')

#Initialization

# a is used in the rekeying phase
a   = randint(0,2**8-1)

key = randint(0,2**8-1)
tv  = randint(0,2**8-1)

for sim in range(2**22):

    # Rekeying (rk =10 or 100)

    for rk in range(100):
            rv  = tv ^ key
            key = (~tv)%255
            tv  = (~rv)%255
            #store rv in a file
            rvs = "%c" % (rv)
            f.write(rvs)

    #rekeying
    b = randint(0,2**8-1)
    c = (b & 0xe0) | (a & 0x1f)
    cs = "%c" % (c)
    nrk= crc.table_driven(cs)
    key = nrk ^ key

# - End Program -
```