Hindawi Publishing Corporation
Journal of Applied Mathematics
Volume 2014, Article ID 460497, 8 pages
http://dx.doi.org/10.1155/2014/460497

Research Article

A Matrix PRNG with S-Box Output Filtering

Rafael Alvarez and Antonio Zamora

Department of Computer Science and Artificial Intelligence (DCCIA), University of Alicante, Campus de San Vicente, Ap. 99, 03080 Alicante, Spain

Correspondence should be addressed to Rafael Alvarez; ralvarez@dccia.ua.es

Received 15 December 2013; Revised 8 August 2014; Accepted 18 August 2014; Published 8 September 2014

Academic Editor: Hyunsung Kim

We describe a modification to a previously published pseudorandom number generator improving security while maintaining high performance. The proposed generator is based on the powers of a word-packed block upper triangular matrix and it is designed to be fast and easy to implement in software since it mainly involves bitwise operations between machine registers and, in our tests, it presents excellent security and statistical characteristics. The modifications include a new, key-derived s-box based nonlinear output filter and improved seeding and extraction mechanisms. This output filter can also be applied to other generators.

1. Introduction

Most cryptographic protocols require unpredictable quantities; these include keys, prime numbers, and challenge values. If these values were predictable, the security of such systems would be compromised.

The most common way to obtain these values is from pseudorandom sequences. Furthermore, a suitable pseudorandom number generator (PRNG) can be used as the keystream generator in a Vernam stream-cipher scheme (see [1, 2]).

A PRNG is a completely deterministic algorithm; the sequence it generates is a function of its inputs and, unlike a truly random generator, its output can be reproduced. This means that we only need the seed (the input to the PRNG) in order to generate the complete output sequence. The output sequence is much longer than the seed and is, in practice, undistinguishable from a really random sequence.

In security applications we need to produce sequences with large periods, high linear complexities, and good statistical properties and satisfy certain unpredictability criteria. Several statistical suites (see [3–5]) that can reject a sequence as nonrandom are available; they involve testing the frequency of bit patterns, autocorrelation, or linear complexity among other metrics.

Most available cryptographic PRNGs are based on linear feedback shift registers (LFSRs). They are so popular because they can be easily implemented in hardware; they produce sequences of large periods with good statistical properties and have a simple structure that can be analysed easily. LFSRs by themselves are not secure but they are commonly enhanced with other techniques to improve their cryptographic properties.

We propose a modification to a previously published PRNG (see [6]) based on word-packed block upper triangular matrices (BUTM) over $\mathbb{Z}_2$ (see [7]) that improves security introducing a key-derived s-box output filter involving a key-schedule algorithm, as well as enhanced seeding and extraction functions.

Similar to the LFSR case, the generator is comprised of two distinct blocks: a generator component that involves linear operations but is proven to generate sequences of a guaranteed period, perfect linear complexity (unlike LFSRs) and excellent statistical properties; and a nonlinear output filtering component that introduces unpredictability and resistance to common attacks.

One of the main contributions of this paper, together with the seeding and extraction algorithms, is the output filter, which is a new design based on the key scheduling algorithm of RC4 (see [8]) to construct four 8×32 s-boxes; it can be

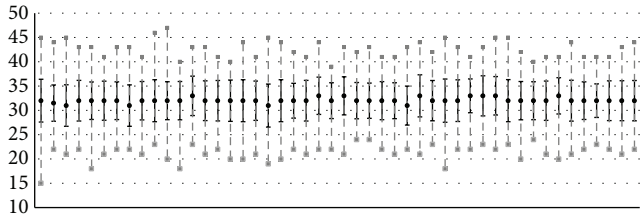

Figure 1: Avalanche analysis of $X^{(t)}$ matrix block.

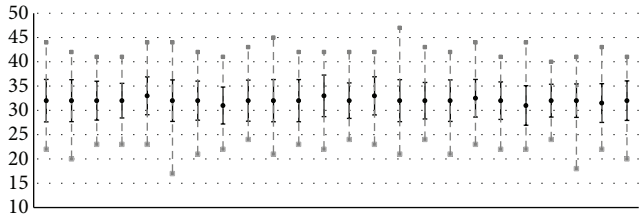

Figure 2: Avalanche analysis of generated sequence.

adapted to other generators, incorporating the concept of key scheduling to PRNGs.

The main key differences and improvements of the proposal regarding the original PRNG (see [6]) are included in the following.

Seed Size. The original PRNG had no minimum seed size specified and a maximum of 3072 bits. The improved PRNG has a minimum of 128 bits of seed specified and up to 2976 (in order to guarantee a valid seed).

Seeding. There was no seeding algorithm specified in the original PRNG; the improved PRNG specifies a seeding algorithm that is avalanche optimized and guarantees a valid (nonzero) seed.

Output Filter. The original PRNG had an output filter based on simple, nonlinear, Boolean functions; the improved PRNG employs an output filter based on the approximation of a 32×32 bit s-box through the combination of four key-dependent 8×32 bit s-boxes. These are indexed by one 32 bit word from the linear component of the PRNG and further combined with an addition modulo 2^{32} operation with another 32 bit word from the linear component.

Key Scheduling. The original PRNG had no key-scheduling algorithm; the improved PRNG specifies a key scheduling algorithm based on the construction of four 8×32 bit key-dependent s-boxes. The fact that these are key-dependent further improves the unpredictability of the proposed output filter.

Blank Rounds. The original PRNG specified no blank rounds; the improved PRNG specifies 64 blank rounds, skipping a total of 98304 bits before producing any output. This further improves the statistical, avalanche, and security properties.

The paper is divided as follows: a description of the generator is given in Section 2.1, packed matrices and related operations are detailed in Section 2.2, while Section 2.3 describes the key scheduling, seeding, and filtering algorithms; we analyse the randomness, avalanche, security, and performance characteristics of the proposal in Section 3; and, finally, some conclusions are given in Section 4.

2. Description

2.1. Generator. Our generator is based on the powers of a BUTM defined over $\mathbb{Z}_p$, with p prime [9]. As we take the different powers of a BUTM, we have as a result a series of matrices that have interesting randomness properties and very long period. Subsequently, each element of the series (each BUTM) can be processed to obtain an output sequence with good statistical values.

Consider the BUTM M defined as

$$M = \begin{bmatrix} A & X \\ O & B \end{bmatrix}, \tag{1}$$

whose entries lie in $\mathbb{Z}_p$, where A is an $r \times r$ matrix, B is an $s \times s$ matrix, X is an $r \times s$ matrix, and O denotes the $s \times r$ zero matrix.

The following result, which forms the basis of the generator, establishes the expression of the different powers of matrix M. It also defines matrix $X^{(h)}$ in terms of A, B, and X.

Theorem 1. *Let M be the BUTM given in* (1). *For any nonnegative integer, h, then,*

$$M^h = \begin{bmatrix} A^h & X^{(h)} \\ O & B^h \end{bmatrix}, \tag{2}$$

where

$$X^{(h)} = \begin{cases} O & si\ h = 0, \\ \displaystyle\sum_{i=1}^{h} A^{h-i} X B^{i-1} & si\ h \geq 1. \end{cases} \tag{3}$$

Also, if $0 \leq t \leq h$, then

$$X^{(h)} = A^t X^{(h-t)} + X^{(t)} B^{h-t}. \tag{4}$$

In order to generate the pseudorandom bit sequence, matrices A and B are predetermined and matrix X is randomly chosen, becoming the seed of the sequence. Then, taking expression (4) we obtain the following series of matrices:

$$X^{(2)}, X^{(3)}, X^{(4)}, \ldots. \tag{5}$$

For each matrix $X^{(h)}$ a bit extraction operation is determined obtaining a sequence of bits like

$$b^2, b^3, b^4, \ldots. \tag{6}$$

The key for obtaining long periods for the sequence given by (6) is constructing matrices A and B as companion matrices to primitive polynomials so the period can be guaranteed to be at least

$$\operatorname{lcm}\left(p^r - 1, p^s - 1\right). \tag{7}$$

The value of p or the sizes of A and B need not be very large in order to achieve long periods. For more information see [6, 10, 11].

Although any prime can be chosen for the generator, we choose $p = 2$ for performance reasons, as highlighted in the following section.

The initial contents of matrices A and B are shown in Algorithms 5 and 6, respectively. These are constructed as companion matrices to primitive polynomials in $\mathbb{Z}_2$, as described before, so the period can be guaranteed to be at least

$$\operatorname{lcm}\left(2^{64}-1, 2^{48}-1\right). \tag{8}$$

2.2. Packed Matrices. The concept of word-packed matrices is essential for the optimized implementation of the generator over $\mathbb{Z}_2$. Word-packed matrices enable adding and multiplying binary matrices just by performing binary operations between processor registers, which is very efficient. For more information see [7].

We define a matrix, whose elements lie in $\mathbb{Z}_2$, as a word-packed matrix if one of its dimensions (rows or columns) is packed as word sized groups of bits.

Operations involving word-packed matrices are equivalent to those between conventional matrices since packed matrices are, essentially, just a way of storing the elements of the matrix so that the computations required can be efficiently implemented as binary operations between processor registers. Nevertheless, they present certain peculiarities of their own that must be taken into account.

The addition of word-packed matrices must be done between matrices of the same type, observing packing orientation: rows or columns. Although they could be unpacked and operated normally, the optimal way is to perform a XOR operation word by word.

The product operation between packed matrices is a little more complex than the addition. The product must be done between matrices of different types and with compatible sizes. The multiplicand has to be a row packed matrix, while the matrix corresponding to the multiplier must be packed by columns.

2.2.1. Matrices and Packness. In order to implement the generator over $\mathbb{Z}_2$, we can see that taking $t = 1$ in expression (4) leaves us with

$$X^{(h)} = AX^{(h-1)} + XB^{h-1}, \tag{9}$$

and the following operations have to be performed per iteration:

$$\begin{aligned} E &= AX^{(h-1)} + XB^{h-1}, \\ X^{(h)} &= E, \\ F &= BB^{h-1}, \\ B^h &= F. \end{aligned} \tag{10}$$

It can be observed that the computations required on each iteration imply that matrices X, B, and their powers $X^{(h)}$ and B^h, must be kept in memory at the same time. It is also necessary to employ temporary matrices E and F since the same matrix cannot simultaneously be source and destination.

Table 1: Different orders of M with $p = 2$.

r	s	Digits
15	8	06
31	8	11
47	8	16
23	16	11
31	16	14
47	16	18
47	32	23
63	32	28
64	48	33
80	48	38
95	48	43
96	53	44

Taking into account the peculiarities of the product operation between packed matrices, we can identify the following matrices and types:

(i) A row packed matrix,

(ii) B row packed matrix,

(iii) B^h column packed matrix,

(iv) X row packed matrix,

(v) $X^{(h)}$ column packed matrix,

(vi) E, F column packed matrices (temporary).

Although the product operation between word-packed matrices generates sparse bits instead of words, these bits can be repacked into the desired format (rows or columns) without a significant performance penalty.

2.2.2. Parameters. Besides determining the format for each matrix, their sizes must also be decided for the correct operation of the implementation.

Several sizes and the number of decimal digits of the corresponding period are shown in Table 1. The option that appears to be more adequate is the $r = 64$, $s = 48$ since it is compatible with usual processor register sizes (32 or 64 bits). Moreover, the order obtained is very high.

2.3. Extraction Mechanism. In this section, we describe the different algorithms that perform seeding, bit extraction, and output filtering using a suitable pseudocode notation involving the following operators:

$\hat{}$ bitwise XOR,

& bitwise AND,

$\gg$ bitwise right shift,

$\ll$ bitwise left shift,

```
for i:= 0 to 255 {
  s[i]:= i
  S0[i]:= 0
  S1[i]:= 0
  S2[i]:= 0
  S3[i]:= 0
}
```

Algorithm 1

```
for c:= 0 to 3 {
  j:= 0
  for i:= 0 to 255 {
    j := (j + s[i] + key[i % 16]) % 256
    t:= s[i]
    s[i]:= s[j]
    s[j]:= t
  }
  for i:= 0 to 255
    S0[i]:= (S0[i]<<8) ^ s[i]
}
```

Algorithm 2

% modulus,

:= assignment,

+ addition modulo 2^{32},

$*$ product.

The generator takes a 128 bit seed (or key) as input and uses it to generate the initial state of block X (Section 2.3.2) and the s-boxes (Section 2.3.1) that conform the output filter (Section 2.3.3). It is trivial to modify the generator to accept longer (or shorter) seeds.

2.3.1. S-Box Construction. The four 8×32 s-boxes are constructed following a similar scheme to the one employed on the RC4 key set-up algorithm (see [8]).

We initialise a temporary 8×8 s-box, s, and clear the final 8×32 s-boxes, $S0, S1, S2, S3$ (see Algorithm 1).

Then, for each s-box ($S0$ shown as an example), we perform 256 substitutions on s involving the key (seed) in the process. This creates an 8×8 s-box, so it is repeated four times and the results are concatenated to form an 8×32 s-box (see Algorithm 2).

The first s-box ($S0$) starts from the initialized s, and each subsequent substitution process builds on the previous ones using the current state of s. This is repeated for the remaining 8×32 s-boxes, $S1, S2, S3$, constructing a total of 16 8×8 s-boxes in the process.

Note that all s-boxes constructed in this way are balanced and have properties similar to purely random, non-key-dependent, s-boxes (see [12] for more information).

```
offs:= 0
for r = 0 to 63 {
  if r is 0 or r is 63 {
    v:= 0x55AA55AA55AA55AA
  }
  else {
    v:= S0[(r + key[offs % 16]) % 256]
    offs:= offs + 1
    v:= v ^ S1[(offs + key[offs % 16]) % 256]
    offs:= offs + 1
    v:= v << 32
    v:= v ^ S2[(r + key[offs % 16]) % 256]
    offs:= offs + 1
    v:= v ^ S3[(offs + key[offs % 16]) % 256]
    offs:= offs + 1
  }
  for c:= 0 to 47
    X[r*48+c]:= (v>>c) % 2
}
```

Algorithm 3

2.3.2. Seeding. The generator is seeded using the X block. It is configured as a conventional binary 64×48 matrix which is then converted to the row and column word packed representations needed by the algorithm. The contents of this block are dependent on the 128 bit seed input.

Each row of this matrix is filled up using a 64 bit word, v, that uses the output of all four s-boxes generated in the previous step, guaranteeing excellent avalanche characteristics (see Section 3). Only the least significant 48 bits of v are actually used.

Rows 0 and 63 are fixed to a specific bit pattern, preventing a full zero X block regardless of the key employed (see Algorithm 3).

The generator is then iterated 64 times without generating any output, further improving avalanche characteristics and overall security.

2.3.3. Filtering and Extraction. The output filtering and extraction mechanism employs two adjacent 32 bit words from matrix $X^{(h)}$; separating the first word, $Xh[c]$, into four bytes, $b0$ to $b3$, which serve as indexes into $S0$ to $S3$, respectively. These four 32 bit words are XORed again and the second word from $X^{(h)}$, $Xh[c]$, is added modulo 2^{32} obtaining a single 32 bit word of output.

This process is repeated to process all 96 32 bit words produced on each iteration of the generator, thus creating 48 32 bit words (1536 bits) of filtered output sequence per iteration.

The array Xh corresponds to the $X^{(h)}$ block associated to iteration h (see Section 2.2.1) (see Algorithm 4).

3. Results

3.1. Randomness Analysis. The resulting generator has been tested successfully with three different statistical suites.

```
offr:= 0
for c:= 0 to 94 increment 2 {
  b0:= Xh[c] % 256
  b1:= Xh[c]>>8 % 256
  b2:= Xh[c]>>16 % 256
  b3:= Xh[c]>>24 % 256
  Seq[offr]:= (S0[b0] ^ S1[b1] ^ S2[b2] ^ S3[b3])
  Seq[offr]:= Seq[offr] + Xh[c + 1]
  offr:= offr + 1
}
```

Algorithm 4

```
u32 A [128]
  := {0x40000000, 0x00000000, 0x20000000, 0x00000000, 0x10000000, 0x00000000,
      0x08000000, 0x00000000, 0x04000000, 0x00000000, 0x02000000, 0x00000000, 0x01000000,
      0x00000000, 0x00800000, 0x00000000, 0x00400000, 0x00000000, 0x00200000, 0x00000000,
      0x00100000, 0x00000000, 0x00080000, 0x00000000, 0x00040000, 0x00000000, 0x00020000,
      0x00000000, 0x00010000, 0x00000000, 0x00008000, 0x00000000, 0x00004000, 0x00000000,
      0x00002000, 0x00000000, 0x00001000, 0x00000000, 0x00000800, 0x00000000, 0x00000400,
      0x00000000, 0x00000200, 0x00000000, 0x00000100, 0x00000000, 0x00000080, 0x00000000,
      0x00000040, 0x00000000, 0x00000020, 0x00000000, 0x00000010, 0x00000000, 0x00000008,
      0x00000000, 0x00000004, 0x00000000, 0x00000002, 0x00000000, 0x00000001, 0x00000000,
      0x00000000, 0x80000000, 0x00000000, 0x40000000, 0x00000000, 0x20000000, 0x00000000,
      0x10000000, 0x00000000, 0x08000000, 0x00000000, 0x04000000, 0x00000000, 0x02000000,
      0x00000000, 0x01000000, 0x00000000, 0x00800000, 0x00000000, 0x00400000, 0x00000000,
      0x00200000, 0x00000000, 0x00100000, 0x00000000, 0x00080000, 0x00000000, 0x00040000,
      0x00000000, 0x00020000, 0x00000000, 0x00010000, 0x00000000, 0x00008000, 0x00000000,
      0x00004000, 0x00000000, 0x00002000, 0x00000000, 0x00001000, 0x00000000, 0x00000800,
      0x00000000, 0x00000400, 0x00000000, 0x00000200, 0x00000000, 0x00000100, 0x00000000,
      0x00000080, 0x00000000, 0x00000040, 0x00000000, 0x00000020, 0x00000000, 0x00000010,
      0x00000000, 0x00000008, 0x00000000, 0x00000004, 0x00000000, 0x00000002, 0x00000000,
      0x00000001, 0xD8000000, 0x00000000};
```

Algorithm 5: Initial contents for matrix A.

```
u32 B [96]
  := {0x40000000, 0x00000000, 0x20000000, 0x00000000, 0x10000000, 0x00000000,
      0x08000000, 0x00000000, 0x04000000, 0x00000000, 0x02000000, 0x00000000, 0x01000000,
      0x00000000, 0x00800000, 0x00000000, 0x00400000, 0x00000000, 0x00200000, 0x00000000,
      0x00100000, 0x00000000, 0x00080000, 0x00000000, 0x00040000, 0x00000000, 0x00020000,
      0x00000000, 0x00010000, 0x00000000, 0x00008000, 0x00000000, 0x00004000, 0x00000000,
      0x00002000, 0x00000000, 0x00001000, 0x00000000, 0x00000800, 0x00000000, 0x00000400,
      0x00000000, 0x00000200, 0x00000000, 0x00000100, 0x00000000, 0x00000080, 0x00000000,
      0x00000040, 0x00000000, 0x00000020, 0x00000000, 0x00000010, 0x00000000, 0x00000008,
      0x00000000, 0x00000004, 0x00000000, 0x00000002, 0x00000000, 0x00000001, 0x00000000,
      0x00000000, 0x80000000, 0x00000000, 0x40000000, 0x00000000, 0x20000000, 0x00000000,
      0x10000000, 0x00000000, 0x08000000, 0x00000000, 0x04000000, 0x00000000, 0x02000000,
      0x00000000, 0x01000000, 0x00000000, 0x00800000, 0x00000000, 0x00400000, 0x00000000,
      0x00200000, 0x00000000, 0x00100000, 0x00000000, 0x00080000, 0x00000000, 0x00040000,
      0x00000000, 0x00020000, 0x00000000, 0x00010000, 0x89400000, 0x00000000};
```

Algorithm 6: Initial contents for matrix B.

Table 2: RandTest statistical comparison.

	Result	Correction
Frequency	0.7200	2.7060
Serial	2.2407	4.6050
Poker 8	250.4640	284.30
Poker 16	65554	65999
Runs	21.0368	23.5418
Autocorr.	0.8074	1.2820
Linear comp.	10000	≥10000

Table 3: PractRand results for a 64 GB sequence.

Test	Raw	Processed	Evaluation
BCFN(2, 13):!	$R = +0.0$	"pass"	Normal
BCFN(2 + 0, 13 − 0)	$R = +0.1$	$p = 0.480$	Normal
BCFN(2 + 1, 13 − 0)	$R = -3.9$	$p = 0.948$	Normal
BCFN(2 + 2, 13 − 0)	$R = -3.4$	$p = 0.918$	Normal
BCFN(2 + 3, 13 − 0)	$R = -1.2$	$p = 0.692$	Normal
BCFN(2 + 4, 13 − 0)	$R = -0.9$	$p = 0.644$	Normal
BCFN(2 + 5, 13 − 0)	$R = -0.5$	$p = 0.579$	Normal
BCFN(2 + 6, 13 − 0)	$R = -2.3$	$p = 0.828$	Normal
BCFN(2 + 7, 13 − 0)	$R = -2.4$	$p = 0.833$	Normal
BCFN(2 + 8, 13 − 1)	$R = -0.6$	$p = 0.595$	Normal
BCFN(2 + 9, 13 − 1)	$R = -1.4$	$p = 0.717$	Normal
BCFN(2 + 10, 13 − 2)	$R = -2.7$	$p = 0.868$	Normal
BCFN(2 + 11, 13 − 3)	$R = -0.1$	$p = 0.509$	Normal
BCFN(2 + 12, 13 − 3)	$R = +4.7$	$p = 0.033$	Normal
BCFN(2 + 13, 13 − 4)	$R = +1.0$	$p = 0.329$	Normal
BCFN(2 + 14, 13 − 5)	$R = +3.4$	$p = 0.088$	Normal
BCFN(2 + 15, 13 − 5)	$R = +0.8$	$p = 0.348$	Normal
BCFN(2 + 16, 13 − 6)	$R = -2.1$	$p = 0.810$	Normal
BCFN(2 + 17, 13 − 6)	$R = +0.9$	$p = 0.320$	Normal
BCFN(2 + 18, 13 − 7)	$R = -0.4$	$p = 0.517$	Normal
BCFN(2 + 19, 13 − 8)	$R = +6.0$	$p = 0.019$	Normal
DC6-9x1Bytes-1	$R = +1.9$	$p = 0.195$	Normal
Gap-16:!	$R = +0.0$	"pass"	Normal
Gap-16:A	$R = -0.2$	$p = 0.598$	Normal
Gap-16:B	$R = -1.0$	$p = 0.754$	Normal

The first one, shown in Table 2, is a custom suite that checks for bit frequency (frequency), bit pair frequency (Serial), 8 bit and 16 bit pattern frequency (poker 8 and poker 16, resp.), runs (contiguous set bits) and gaps (contiguous unset bits) in the sequence (runs), autocorrelation, and linear complexity. We have made available this suite on GitHub (see [3]). The proposed generator passes all the tests successfully.

The second suite is PractRand (see [4]). We have tested our proposed generator for sequences up to 64 GB (2^{36} bits) in length, passing all tests successfully as shown in Table 3. This also highlights the potential for generating very long, high quality, binary sequences.

The third suite is comprised of the 160 statistics included in TestU01 1.2.3 BigCrush battery (see [5]). The proposed generator passes all tests from this stringent suite; the extensive report is included as supplementry material.

3.2. Avalanche Analysis. Avalanche is a very important characteristic in cryptographic primitives in order to prevent successful cryptanalysis.

It is defined as the number of output bits that change when a single input bit is flipped, and the expected outcome is that roughly half of the output bits change when this happens.

In this case, we have taken 128 different seeds that differ in a single bit and have measured avalanche in two different points: the $X^{(t)}$ block (Figure 1, pre-output-filter) and generated sequence (Figure 2, post-output-filter). The graphs depict the mean difference value, together with one standard deviation above and below the mean value, as well as minimum and maximum values for each 64 bit word in the test sample.

In both cases, the results are excellent with no abnormal values and with most of the population very close to 32, which is the expected value.

3.3. Security

3.3.1. Security Parameters. The proposed generator could be employed as the source for random values, nonces, keys, and so forth, as well as the key stream generator in a Vernam stream cipher.

In this case, it accepts keys of 128 bits in size, but the seeding algorithm (see Section 2.3.2) can be modified to accept longer keys with ease, while maintaining the excellent avalanche characteristics (see Section 3.2).

The maximum capacity of the X block is $62 \times 48 = 2976$ bits, since the first and last rows are fixed to a specific pattern to guarantee a nonzero X block.

3.3.2. Linear Generator. The generator component is based on the powers of a 2×2 BUTM and has excellent statistical characteristics, passing all tests of stringent suites (see Section 3.1).

It also guarantees a period of at least

$$\operatorname{lcm}\left(2^{64} - 1, 2^{48} - 1\right), \tag{11}$$

that is the period of matrix M since blocks A and B are constructed as companion matrices of primitive polynomials in $\mathbb{Z}_2$. The actual expected period is that of matrix M times the number of bits produced per iteration:

$$\operatorname{lcm}\left(2^{64} - 1, 2^{48} - 1\right) \times 1536. \tag{12}$$

Regarding linear complexity, the generated sequences present the expected linear complexity of a random sequence (half the sequence length, see Table 2). This implies that the generator cannot be reduced to a simple LFSR.

The generator is a linear algorithm, involving exclusively addition and multiplication operations over $\mathbb{Z}_2$. This means that the generator by itself is susceptible to linear cryptanalysis where a linear equation system can be set up in order

to obtain the seed state (block X) after a certain number of iterations. This type of attack is prevented by the nonlinear filter component.

3.3.3. Nonlinear Filter. Substitution boxes (or s-boxes) are simple substitution tables where an input value is transformed into a different output value; the most common are 8×8 bits (byte as input and output) and 8×32 bits (byte as input and four byte word as output). They are essential in many cryptosystem designs since they can introduce the required nonlinearity characteristics, making cryptanalysis a more difficult endeavour (see [12–15]).

The proposed nonlinear filter (see Sections 2.3.1 and 2.3.3) employs four 8×32 key-dependent s-boxes and takes a 32 bit word from the generator as input and combines a second word from the generator with the output.

There have been some known attacks on RC4 (see [16–18]); these are mostly based on biases of the first bytes of the output sequence. Although it performs value swapping in similar ways as the key scheduling algorithm of RC4, there are several reasons in which the proposed nonlinear filter differs from it and why we believe these known attacks are not applicable in this case.

(i) *Static s-boxes.* Unlike in RC4, which uses the key scheduling algorithm to determine the initial state of an evolving 8×8 s-box, our proposal uses the four 8×32 s-boxes to filter the output of the linear generator and the s-boxes do not evolve with sequence generation.

(ii) *Sequence is not sourced on s-boxes.* In our proposal, the source of the output sequence is the linear generator component and not the s-box evolution; therefore, the biases associated with the RC4 key scheduling algorithm do not affect the sequences produced by the proposed PRNG.

(iii) *Multiple s-boxes.* The proposed nonlinear filter employs four different 8×32 s-boxes that are combined together to generate the output, unlike in RC4 where a single 8×8 s-box is employed.

(iv) *S-boxes are not output directly.* Furthermore, the output of the four 8×32 s-boxes is combined with a second 32 bit word from the linear generator using addition modulo 2^{32}. This means that, unlike in RC4, the output sequence is never the direct output of the s-boxes.

(v) *Blank rounds.* The seeding algorithm (see Section 2.3.2) specifies that the generator is iterated 64 times (and therefore $1536 \times 64 = 98304$ bits of output are skipped) avoiding the pitfalls of the early biases in RC4.

The proposed nonlinear filter presents defences against common attacks.

(i) *Linear cryptanalysis.* Key-derived s-boxes behave essentially as random s-boxes and have highly nonlinear characteristics (see [12, 13]), together with the fact that the second word from the generator is added modulo 2^{32} (involving operations over $\mathbb{Z}_2$ and $\mathbb{Z}_{2^{32}}$) complicate approximating the whole PRNG with a linear equation system.

(ii) *Differential cryptanalysis.* Although it is not a common attack against stream ciphers, the nonlinearity values of the 16 component 8×8 s-boxes are within secure bounds (see [12, 13]). Furthermore, the fact that the s-boxes are not fixed but key-dependent makes differential cryptanalysis more difficult.

(iii) *Correlation and statistical attacks.* There are several design features to prevent these type of attacks: the 64 blank rounds in the seeding algorithm, only 48 32 bit words are output per iteration from 96 words possible, one word from the generator is used as the input to the s-boxes while the adjacent one is combined with the output of the s-boxes using a different operation. Moreover, the excellent statistical characteristics of the linear generator component help prevent biases and other problems.

Table 4: Performance benchmark.

	MB/s
Proposed PRNG	133.1
AES-256 (OFB)	71.3
RC4	218.1
Salsa20	209.7
HC128	176.7

3.4. Performance. We have included a performance benchmark in Table 4. This table includes speed measurements for common algorithms, such as AES with a 256 bit key in output feedback mode (OFB) (see [19]), the RC4 stream cipher (see [8]), and the Salsa20 (see [20]) stream cipher and the HC128 stream cipher (see [21]), together with our proposal.

All algorithm implementations are single thread, pure native compiler-optimized code, without hardware acceleration or processor-specific high-performance instruction set extensions.

Although not as fast as other lighter-weight stream ciphers, the proposed generator achieves acceptable performance and presents valuable characteristics.

One of them is that, being a matrix based generator, it is essentially an *embarrisingly parallel* problem and, therefore, capable of taking advantage of the multiple core architecture of modern CPUs or vector processing in GPUs.

Another advantage comes from the fact that the whole matrix computations involve binary operations between registers, and matrix sizes can be adjusted to profit from architectures that have bigger register sizes. The proposed design employs a 64×48 bit matrix size, taking advantage of current 64 bit architectures, but it is trivial to adjust this size for maximum performance on future architectures if necessary.

4. Conclusions

We have presented a modification on a block matrix PRNG introducing a key-dependent s-box output filter as well as different seeding and initialization algorithms to improve security and nonlinearity. It employs a word packing technique in order to optimise computations over $\mathbb{Z}_2$ reducing them to fast, native, register bitwise operations. Additionally, the word packing system can directly take advantage of more powerful processors with bigger registers.

The resulting generator produces sequences of great quality in terms of randomness, passing battery tests like PractRand [4] or TestU01 [5], and with a very long period. It is also capable of accepting very large keys if necessary.

The key-dependent s-box output filter is the result of adapting the concept of block cipher key-scheduling to a key-stream generator. It offloads some computations to each seed change cycle while maintaining high performance during sequence generation. An additional benefit is that it multiplies the cost of brute force key search. Although somewhat based on the key scheduling algorithm of RC4 [8], it is a new design generating four 8×32 key-dependent s-boxes that do not evolve during sequence generation.

Possible future work includes parallel implementations of the proposed generator on suitable CPU and GPU platforms, assembler optimization, and other performance analyses, as well as the adaptation of the nonlinear filter component to other PRNG and further analysis.

Appendix

For more details see supplementry material.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgment

Research partially supported by the Spanish MINECO under Project TIN2011-25452.

References

[1] N. Ferguson, B. Schneier, and T. Kohno, *Cryptography Engineering: Design Principles and Practical Applications*, John Wiley and Sons, New York, NY, USA, 2012.

[2] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 2010.

[3] R. Alvarez, "RandTest," 2004, http://github.com/rias/randtest.

[4] C. Doty-Humphrey, PractRand, 2013, http://pracrand.sourceforge.net/.

[5] P. L' Ecuyer and R. Simard, "TestU01: a C library for empirical testing of random number generators," *Association for Computing Machinery: Transactions on Mathematical Software*, vol. 33, article 22, no. 4, Article ID 1268777, 2007.

[6] R. Alvarez, J. Climent, L. Tortosa, and A. Zamora, "An efficient binary sequence generator with cryptographic applications," *Applied Mathematics and Computation*, vol. 167, no. 1, pp. 16–27, 2005.

[7] R. Alvarez, M. J. Castel, L. Tortosa, and A. Zamora, "Optimizing matrix operations in Z_2 by word packing," *Applied Mathematics Letters*, vol. 22, no. 2, pp. 242–244, 2009.

[8] R. L. Rivest, *The RC4 Encryption Algorithm*, RSA Data Security, 1992.

[9] R. Lidl and H. Niederreiter, *Introduction to Finite Fields and their Applications*, Cambridge University Press, Cambridge, UK, 1st edition, 1994.

[10] R. W. K. Odoni, V. Varadharajan, and P. W. Sanders, "Public key distribution in matrix rings," *Electronics Letters*, vol. 20, no. 9, pp. 386–387, 1984.

[11] V. Varadharajan and R. Odini, "Security of public key dist ribution in matrix rings," *Electronics Letters*, vol. 22, no. 1, pp. 46–47, 1986.

[12] R. Alvarez and A. Zamora, "Randomness analysis of key-derived S-boxes," in *Advances in Intelligent Systems and Computing*, vol. 239, pp. 611–618, Springer, 2014.

[13] R. Alvarez and G. McGuire, "S-Boxes, APN functions and related codes," in *Enhancing Cryptographic Primitives with Techniques from Error Correcting Codes*, vol. 23 of *NATO Science for Peace and Security Series-D: Information and Communication Security*, pp. 49–62, IOS Press, 2009.

[14] I. Hussain, T. Shah, M. A. Gondal, and W. A. Khan, "Construction of cryptographically strong 8×8 S-boxes," *World Applied Sciences Journal*, vol. 13, no. 11, pp. 2389–2395, 2011.

[15] S. Murphy and M. J. B. Robshaw, "Key-dependent S-boxes and differential crypt-analysis," *Designs, Codes and Cryptography*, vol. 27, no. 3, pp. 229–255, 2002.

[16] S. Fluhrer, I. Mantin, and A. Shamir, "Weaknesses in the key scheduling algorithm of RC4," in *Selected Areas in Cryptography*, pp. 1–24, Springer, New York, NY, USA, 2001.

[17] S. S. Gupta, S. Maitra, G. Paul, and S. Sarkar, "(Non-) Random Sequences from (Non-) Random Permutations—analysis of RC4 Stream Cipher," *Journal of Cryptology*, vol. 27, no. 1, pp. 67–108, 2014.

[18] A. Klein, "Attacks on the RC4 stream cipher," *Designs, Codes and Cryptography*, vol. 48, no. 3, pp. 269–286, 2008.

[19] J. Daemen and V. Rijmen, "AES proposal: Rijndael," in *Proceedings of the 1st Advanced Encryption Standard (AES) Conference*, 1998.

[20] D. J. Bernstein, "The Salsa20 family of stream ciphers," in *New Stream Cipher Designs*, vol. 4986 of *Lecture Notes in Computer Science*, pp. 84–97, Springer, Berlin, Germany, 2008.

[21] H. Wu, "The stream cipher HC-128," in *New Stream Cipher Designs*, vol. 4986 of *Lecture Notes in Computer Science*, pp. 39–47, Springer, Berlin, Germany, 2008.