Available online at www.sciencedirect.com
ScienceDirect
Mathematics and Computers in Simulation 143 (2018) 114–124
MATHEMATICS AND COMPUTERS IN SIMULATION
www.elsevier.com/locate/matcom

Original articles

Efficient deterministic and non-deterministic pseudorandom number generation

Jie Li[a,*], Jianliang Zheng[b], Paula Whitlock[c]

[a] *Department of Computer Systems, Farmingdale State College, SUNY, Farmingdale, NY 11735, USA*
[b] *Independent Scholar, New York, NY, USA*
[c] *Department of Computer and Information Science, Brooklyn College, CUNY, New York, NY 11210, USA*

Received 3 January 2016; accepted 22 July 2016
Available online 3 August 2016

Abstract

A high performance and high quality pseudorandom number generator is presented in this paper. It takes less than one clock cycle to generate a pseudorandom byte on an Intel core i3 processor and passes all the 6 TestU01 batteries of tests. The generator can work in either deterministic mode or non-deterministic mode. When working in deterministic mode, it can be used for high speed data encryption and in other applications that require deterministic and reproducible pseudorandom sequences. When working in non-deterministic mode, the generator behaves much like a true random number generator, but with the advantages of low cost, high performance, and general availability. It is good for many applications that currently rely on true random number generators

Keywords: Pseudorandom number generator; Non-deterministic; Randomness; Performance; Cryptanalysis

1. Introduction

A pseudorandom number generator (PRNG) usually generates pseudorandom numbers in a deterministic way. This makes it possible to reproduce a pseudorandom sequence, which is necessary or useful in some applications, such as data encryption and modeling. A true random number generator (TRNG), on the other hand, works in a non-deterministic way. Non-deterministic random number generation is preferred in applications such as gambling and lottery, where fairness is essential and manipulation should not be possible. TRNGs, however, are still not widely adopted for several reasons: too expensive; relatively slow (although high speed TRNGs such as quantum random number generators can be used, please see http://qrng.physik.hu-berlin.de/); and not generally available.

In this paper we propose a high performance and high quality pseudorandom number generator. This new generator, referred to as MaD1, is mainly designed for secure applications, but can also be used in non-cryptographic applications. MaD1 uses a high quality byte-oriented pseudorandom permutation function for key scheduling and state initialization. This renders an excellent avalanche effect that is comparable to those of standard hash functions

* Corresponding author.
E-mail addresses: jie.li@farmingdale.edu (J. Li), zheng@ee.ccny.cuny.edu (J. Zheng), whitlock@sci.brooklyn.cuny.edu (P. Whitlock).

http://dx.doi.org/10.1016/j.matcom.2016.07.011

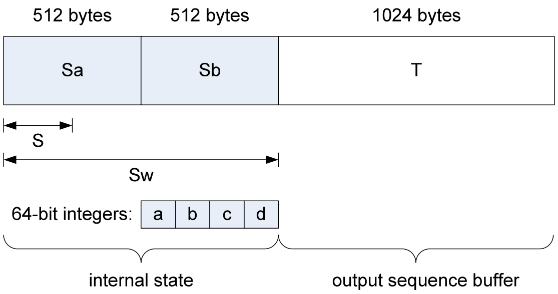

Fig. 2.1. Data structure.

and helps the generator to withstand a large number of attacks. MaD1 demonstrates excellent statistical property and clears all the 6 TestU01 batteries of tests [11]. It takes the advantages of modern 64-bit platforms and uses an integer-oriented pseudorandom mapping function for state transition and pseudorandom number generation. On an Intel core i3 processor, the generator can reach a speed close to 0.6 clock cycle per byte.

Besides deterministic pseudorandom number generation, MaD1 can also work in non-deterministic mode. In this mode, the generator behaves much like a true random number generator by periodically querying some non-deterministic random source and using it as an unpredictable entropy input. Running in this mode has virtually no impact on the cost or performance. Nor does it affect the availability or usability of the generator since no dedicated device or special setup is needed. Many applications currently relying on true random number generators can benefit from this mode.

The rest of this paper is structured as follows. In Section 2, we describe the algorithm in detail. Next, in Section 3, we present the security analysis. Then, in Sections 4 and 5, we give the statistical testing results and performance testing results respectively. Finally, we conclude in Section 6.

2. Algorithm details

In this section we present the algorithm details. All components will be described using pseudo code that largely follows the conventions of C/C++ programming language. Hexadecimal numbers are prefixed by "0x" and all variables and constants are unsigned integers in little endian.

2.1. MARC-bb

In MaD1, we use an iteration-reduced version of MARC [25], referred to as MARC-bb (bb stands for building block), for key scheduling and state initialization. The only difference between MARC-bb and MARC is that the MARC-bb key scheduling algorithm (KSA) iterates 320 times while the MARC KSA iterates 576 times to shuffle the identity permutation table. The reason we choose MARC-bb over MARC is that the MARC-bb KSA already has an avalanche effect comparable to that of standard hash functions (see comparison results in [14]). The additional 256 iterations implemented in MARC KSA are mainly for increasing the security margin, which is achieved in MaD1 through the additional shuffling introduced after the key scheduling. The complete MARC-bb algorithm is given in Listing 1.

2.2. Data structure

MaD1 maintains a data structure shown in Fig. 2.1, which comprises two 512-byte state tables, denoted as *Sa* and *Sb*, four 64-bit integers, denoted as *a*, *b*, *c*, and *d*, and one 1024-byte output sequence buffer, denoted as *T*. The two state tables *Sa* and *Sb* and the four 64-bit integers *a*, *b*, *c*, and *d* construct the internal state. The output sequence buffer *T* is used for buffering pseudorandom numbers generated from the internal state. The first 256 bytes of *Sa* are also referred to as state table *S* to indicate that they play the role of the permutation table *S* in key scheduling. The concatenation of *Sa* and *Sb* is sometimes used as a large 1024-byte state table, referred to as *Sw*.