Efficient Deterministic and Non-Deterministic Pseudorandom Number Generation

Jie Li, Jianliang Zheng, Paula Whitlock

Outline

- Introduction
- MaD1 Algorithm
  - A Building Block: MARC-bb
  - Data Structure
  - Key Scheduling
  - Initialization of Internal State
  - Deterministic Pseudorandom Generation
  - Non-Deterministic Pseudorandom Generation
- Security Analysis
- Statistical Test
- Performance Test
- Summary

Introduction

- MaD family of cryptographic and pseudorandom number generators:
  - MaD0 – pseudorandom number generator
  - MaD1 – produces both cryptographic and pseudorandom streams
  - Mad2/3 – produces most secure cryptographic cipher stream
- MaD family evolved out of an attempt to improve the RC4 stream cipher called MARC (modified ARC – open source version of RC4)
- MARC was designed to resist well-known security attacks on RC4
- Mad1 is discussed here. It sacrifices some security features to provide high speed generation. It can be used both as a deterministic or non-deterministic generator.

MARC-bb: MARC as a Building Block

- MARC is a byte-oriented PRNG, not very fast.
- MARC-bb is a building block for advanced PRNGs.
- MARC-bb reduce the iterations in the key scheduling algorithm from 576 used in MARC to 320.
- It has the same state transition function as MARC
- MARC-bb has an avalanche effect property comparable to hash functions. It satisfies the strict avalanche criterion.

MARC-bb **Key Scheduling Algorithm (KSA)**

```
## addition (+) and increment (++) operations   ##
## are performed modulo 256; except variable r, ##
## which is a 16-bit unsigned integer, all other##
## variables are 8-bit unsigned integers.       ##
## % means modulo; ^ means bitwise XOR.         ##

# Key Scheduling Algorithm (KSA)
for i from 0 to 255
   S[i] = i
endfor
i = 0
j = 0
k = 0
for r from 0 to 319
   j = j + S[i] + key[r % keylength]
   k = k ^ j
   left_rotate(S[i], S[j], S[k])
   i++
endfor
```

MARC-bb **Pseudorandom Generation Algorithm(PRGA)**

```
# Pseudorandom Generation Algorithm (PRGA)
# (j and k are from KSA)

i = j + k
while GeneratingOutput
  i++
  j = j + S[i]
  k = k ^ j
  swap(S[i], S[j])
  m = S[j] + S[k]
  n = S[i] + S[j]
  output S[m]
  output S[n]
  output S[m ^ j]
  output S[n ^ k]
endwhile
```

MARC-bb: Chi-Square Statistic Test

- Flip one input bit each time and compare the initialized state s' with the initialized state s before flipping.
- Compute the Hamming distance between s' and s → number of output bits changed.
- Compute the chi-square value

$$\chi^2 = \sum_{m=0}^{L} \frac{(O_m - E_m)^2}{E_m}$$

Om = the actual number of times that exactly *m* output bits are flipped in N experiments

Em = the expected number of times that *m* output bits are flipped for a binomial distribution

L = the bit length of the output

- Compare with the critical value (C.V.) at α = 0.01.
- If $\chi 2$ > C.V., reject *H*0: observed distribution matches a binomial distribution; Otherwise, accept *H*0 .

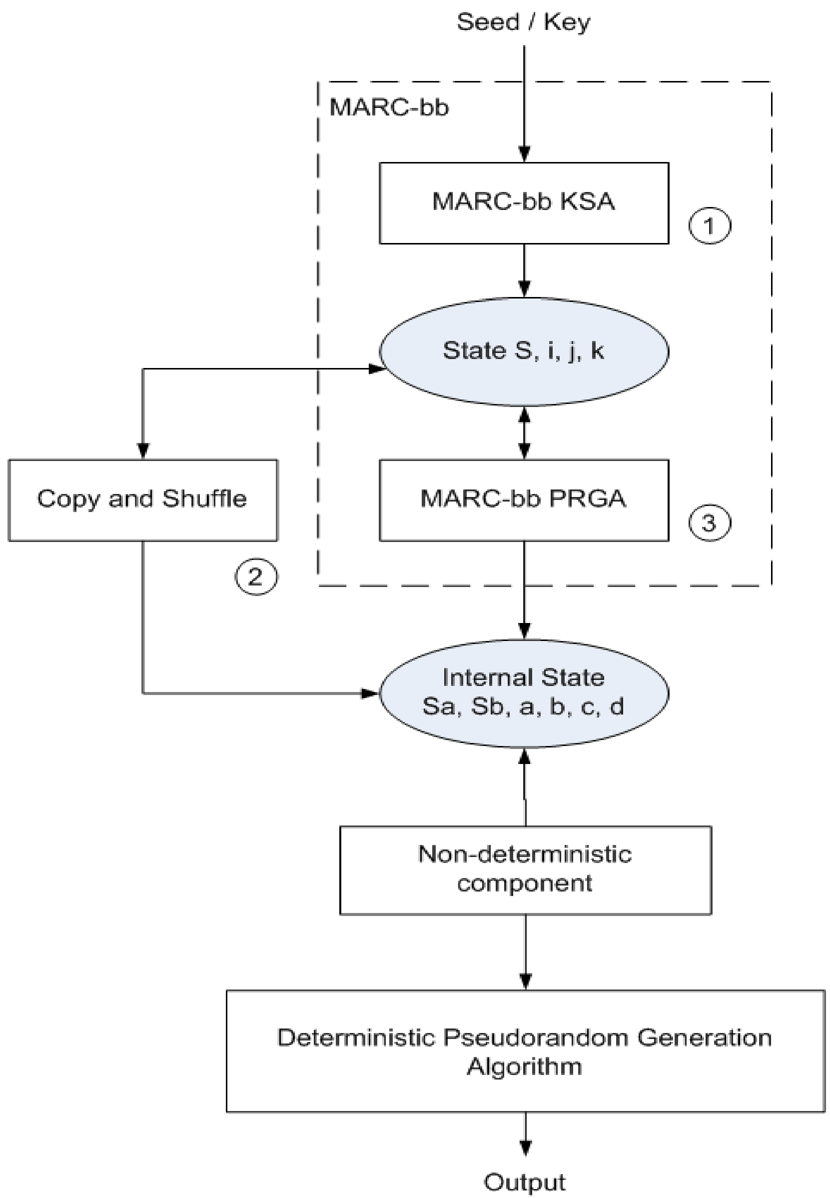
MARC-bb: Chi-Square Statistic Test (Cont.)

Algorithm	Input size (bytes)	Output size (bytes)	d.o.f.	χ2	C.V. (α=0.01	Reject H 0?
MD5	64	16	128	49.527	168.233	No
SHA1		20	160	66.401	204.633	No
SHA2		32	256	77.629	311.674	No
MARC-bb		32	256	79.46	311.674	No
		256*	2047	238.36	2199.06	No
RC4		256*	2047	4.56×10^{55}	2199.06	Yes
RC4 (+64 iterations)		256*	2047	1.87×10^{16}	2199.06	Yes
RC4 (+256 iterations)		256*	2047	244.29	2199.06	No

N = 100352 experiments

- MARC-bb KSA has a similar avalanche effect as standard hash algorithms.
- More shuffling helps to improve the avalanche effect of RC4 KSA.

MaD1 – An Ultrafast High Quality PRNG



MaD1 Model

Key scheduling

Initiali-zation

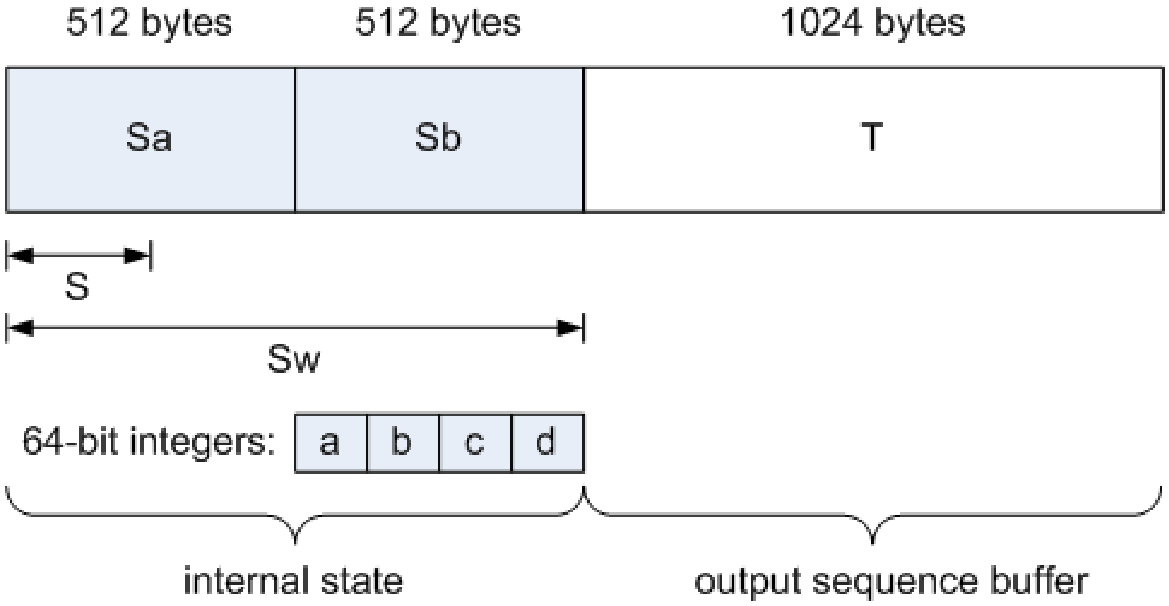
Pseudo random genera-tion

MaD1 – Algorithm Design 1

- Data Structure (next slide)
- Key scheduling
  - Key size: up to 64 bytes (512 bits)
  - MARC-bb KSA
- Initialization
  - state S (the first 256 bytes of Sa) is initialized using MARC-bb KSA
  - The second 256 bytes of Sa and 512 bytes of Sb are initialized using copy-and-shuffle process.
  - Four integers a, b, c, and d are initialized using MARC-bb PRGA.
- Pseudorandom generation
  - Use 64-bit operations -- All state tables (Sa and Sb) and output sequence buffer T are cast into and used as 64-bit integer arrays.
  - Each generation round consists of 32 iterations.
  - In each iteration, two 64-bit integers are generated and one 64-bit integer element of state table S is updated.

MaD1 – Algorithm Design (cont.)

Data Structure



Initialization: copy-and-shuffle function

```
## State table S and index i, j, and k are initialized
    using MARC-bb KSA.
## addition (+) and increment (++) operations  are
    performed modulo 256


for r from 0 to 255
    i++
    j = j + S[i]
    k = k ^ j
    left_rotate(S[i], S[j], S[k])
Endfor


Note:  left_rotate(s[i], s[j], s[k]) means
       tmp=s[i], s[i]=s[j], s[j]=s[k], s[k]=tmp
```

Pseudorandom Generation Algorithm

```
## additions are performed modulo 0x10000000000000000;  ##
## & means bitwise AND; | means bitwise OR;    ##
## << means bitwise logical left shift;       ##
## >> means bitwise logical right shift.      ##

# declare a byte array of size 64
byte x[64]

# cast the byte array into 64-bit integer array
x[64] => x64[8]
```

Pseudorandom Generation Algorithm (cont.)

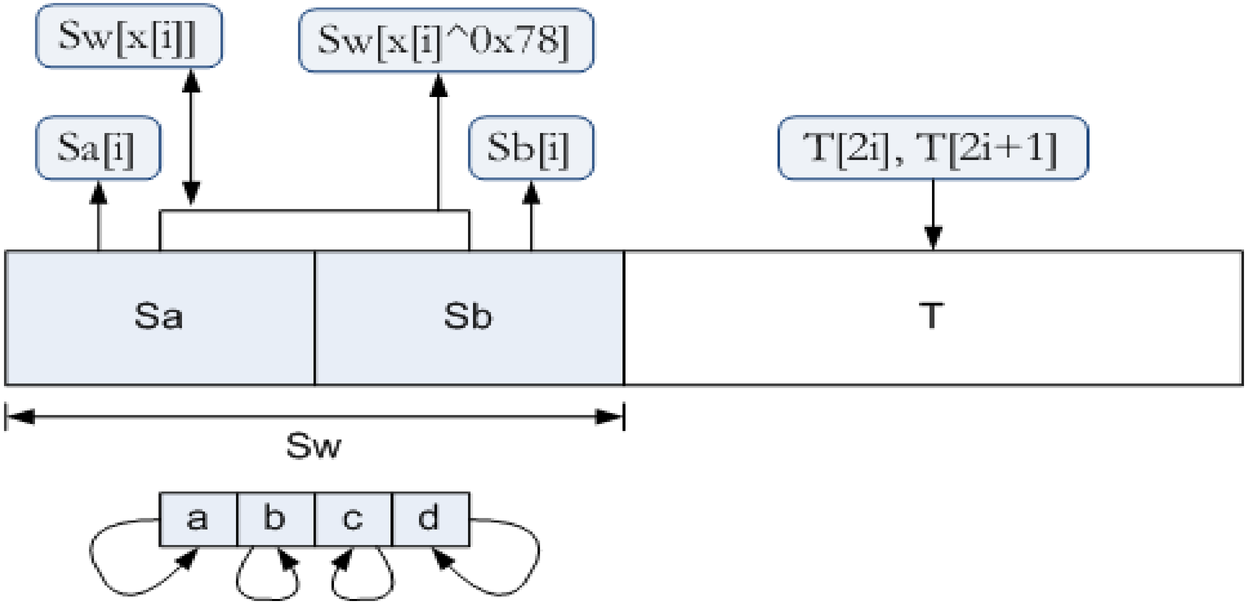
```
# populate array x (through x64)
M = 0x7878787878787878
N = 0x0405060700010203
x64[0] = (a & M) | N
x64[1] = (b & M) | N
x64[2] = (c & M) | N
x64[3] = (d & M) | N
x64[4] = ((a >> 1) & M) | N
x64[5] = ((b >> 1) & M) | N
x64[6] = ((c >> 1) & M) | N
x64[7] = ((d >> 1) & M) | N
```

Pseudorandom Generation Algorithm (cont.)

```
# output and update the internal state
for i from 0 to 63
   a = a << 1
   b = b >> 1
   a = a + Sw[x[i]]
   b = b + Sw[x[i]^0x78]
   c = c + Sa[i]
   d = d + Sb[i]
   T[2i] = c ^ (a + d)
   T[2i+1] = d ^ (b + c)
   Sw[x[i]] = a + b
endfor
```

MaD1 – Algorithm Design 6

- Variable x is a byte array used as indices to access state tables.
- Sw[x[i]] and Sw[x[i]^0x78] introduce pseudorandom indirect access.
- Index i guarantees all state elements get involved in each generation round.
- Sw[x[i]], Sw[x[i]^0x78], Sa[i], and Sb[i] are distinct and different from any of the four state table integers used in the previous or next three iterations.
- In each iteration, two 64-bit integers are generated; Integers a, b, c, d, and a "random" element in Sw are updated.



MaD1 - Period

- MaD1 has an 8448 bit integer-oriented internal state.
- Transition of the integer-oriented state follows a pseudorandom mapping.
- The average period ≈ 2^4224.

MaD1 – Security Analysis

Attacks:

- Correlation attacks, weak keys, related key attacks, etc
- Time-Memory Tradeoff Attacks
- Guessing Attacks
- Algebraic Attacks
- Distinguishing Attacks
- Differential Attacks

Countermeasures in MaD1

- Large internal state
- State initialization with great avalanche property
- Indirect access of state element and special index control
- Non-linear pseudorandom generation
- Pseudorandom mapping state transition

NDPRNG: Non-Deterministic Pseudorandom Number Generation

- Non-deterministic random number generation is preferred in some applications.
  - key/seed generation
  - gambling and lottery
- Existing solutions
  - TRNGs:
    - expensive
    - relatively slow
    - not generally available.
  - PRNGs with entropy inputs:
    - often using cryptographic primitives
    - complicated algorithm and slow speed

NDPRNG - Design Goal and Approach

- Introduce non-deterministic feature into deterministic generator without affecting other features.
- Focus on non-deterministic feature only.
  - leaving randomness, security, etc. to deterministic algorithm
- Maintain the availability of the generators.
  - using generally available entropies only
- Minimize the impact on performance.
  - using as less entropy inputs as possible
  - not using special entropy accumulation, evaluation, processing, and distribution methods

NDPRNG - Entropy Selection

- Commonly used entropies
  - user interactions with the machine
  - hard drive latency
  - disk timings and interrupt timings
  - CPU cycle count and jiffies count
  - number of threads/processes
  - memory/disk utilization and other system information
- Our choice: CPU cycle count
  - available on most processors
  - accessible from any program (not only from the kernel)
  - changing at a relatively high rate
  - low cost
  - difficult to manipulate or predict

MaD1-Non-Deterministic Pseudorandom Generation Algorithm

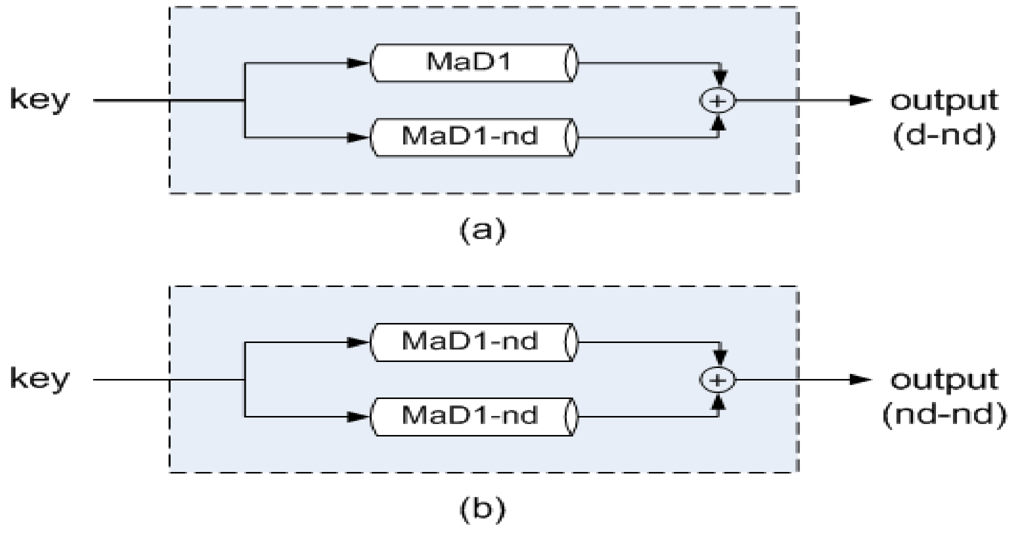
Non-Deterministic Pseudorandom Generation Algorithm

```
# read CPU cycle count
e = readCCC();

# preprocess the cycle count
e = e + (e << 7);
e = e + (e << 19);
e = e + (e << 37);

# use the preprocessed value to modify a, b, c, and d
a = a ^ e;
b = b ^ e;
c = c ^ e;
d = d ^ e;

# continue with the deterministic PRGA
```

NDPRNG – Overall Effects of Algorithm Modification

Property	**Impact**
Randomness	Positive
Security	Positive
Period	Positive
Performance	Negative, but trivial
Availability	Same
Ease of use	Same
Cost	Same
Non-deterministic feature	Added

MaD1 - Statistical Test



Battery	Parameters	NoP	Failures			
			d	nd	d-nd	nd-nd
SmallCrush	Built-in	15	0	0	0	0
Crush	235 random numbers	144	0	0	0	0
BigCrush	238 random numbers	160	0	0	0	0
Rabbit	32x109 bits	40	0	0	0	0
Alphabit	32x109 bits	17	0	0	0	0
BlockAlphabit	32x109 bits	102	0	0	0	0

MaD1 – Performance Test

Pseudorandom Number Generation Speed (cycle/byte)

Generator	Sequence size (KB)					
	1	5	10	100	1000	10000
RC4	9.53	7.67	7.09	6.98	7.04	7.04
HC-128	55.21	13.27	7.96	3.58	3.15	3.11
MaD1 (32-bit)	47.97	12.01	7.09	3.04	2.61	2.59
MaD1 (64-bit)	38.70	8.06	4.28	0.99	0.63	0.61

MaD1 - Summary

- a new word-based pseudorandom number generator
- a huge internal state of 8448 bits, long period
- secure against various known attacks
- Very good statistical properties - passes all TESTU01 tests
- ultrafast
- Non-deterministic feature added with little cost