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.
## Key Scheduling Algorithm (KSA)

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

\[ i = j + k \]
while GeneratingOutput
  \[ i++ \]
  \[ j = j + S[i] \]
  \[ k = k \oplus j \]
  \[ \text{swap}(S[i], S[j]) \]
  \[ m = S[j] + S[k] \]
  \[ n = S[i] + S[j] \]
  \[ \text{output} \ S[m] \]
  \[ \text{output} \ S[m] \]
  \[ \text{output} \ S[n] \]
  \[ \text{output} \ S[n] \]
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
  \[
  \chi^2 = \sum_{m=0}^{L} \frac{(O_m - E_m)^2}{E_m}
  \]
  \( O_m = \) the actual number of times that exactly \( m \) output bits are flipped in \( N \) experiments
  \( E_m = \) 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 \( \alpha = 0.01 \).
- If \( \chi^2 > \text{C.V.} \), reject \( H_0 \): observed distribution matches a binomial distribution; Otherwise, accept \( H_0 \).
### MARC-bb: Chi-Square Statistic Test (Cont.)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Input size (bytes)</th>
<th>Output size (bytes)</th>
<th>d.o.f.</th>
<th>$\chi^2$</th>
<th>C.V. (α=0.01)</th>
<th>Reject $H_0$?</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD5</td>
<td></td>
<td>16</td>
<td>128</td>
<td>49.527</td>
<td>168.233</td>
<td>No</td>
</tr>
<tr>
<td>SHA1</td>
<td></td>
<td>20</td>
<td>160</td>
<td>66.401</td>
<td>204.633</td>
<td>No</td>
</tr>
<tr>
<td>SHA2</td>
<td></td>
<td>32</td>
<td>256</td>
<td>77.629</td>
<td>311.674</td>
<td>No</td>
</tr>
<tr>
<td>MARC-bb</td>
<td>64</td>
<td>32</td>
<td>256</td>
<td>79.46</td>
<td>311.674</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>256*</td>
<td>2047</td>
<td>238.36</td>
<td>2199.06</td>
<td>No</td>
</tr>
<tr>
<td>RC4</td>
<td></td>
<td>256*</td>
<td>2047</td>
<td>$4.56 \times 10^{55}$</td>
<td>2199.06</td>
<td>Yes</td>
</tr>
<tr>
<td>RC4 (+64 iterations)</td>
<td></td>
<td>256*</td>
<td>2047</td>
<td>$1.87 \times 10^{16}$</td>
<td>2199.06</td>
<td>Yes</td>
</tr>
<tr>
<td>RC4 (+256 iterations)</td>
<td></td>
<td>256*</td>
<td>2047</td>
<td>244.29</td>
<td>2199.06</td>
<td>No</td>
</tr>
</tbody>
</table>

$N = 100352$ 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

Seed / Key

MARC-bb

MARC-bb KSA

State S, i, j, k

Copy and Shuffle

MARC-bb PRGA

Internal State Sa, Sb, a, b, c, d

Non-deterministic component

Deterministic Pseudorandom Generation Algorithm

Output

Key scheduling

Key scheduling

Initialization

Pseudo random generation
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 $S_a$) is initialized using MARC-bb KSA
- The second 256 bytes of $S_a$ and 512 bytes of $S_b$ 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 ($S_a$ and $S_b$) 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.
Data Structure

- Sa
- Sb
- T

512 bytes 512 bytes 1024 bytes

64-bit integers: a b c d

Sw

S

internal state

output sequence buffer
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]
# 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
# 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 \( \approx 2^{4224} \).
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
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
Non-Deterministic Pseudorandom Generation Algorithm

```plaintext
# 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

<table>
<thead>
<tr>
<th>Property</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Randomness</td>
<td>Positive</td>
</tr>
<tr>
<td>Security</td>
<td>Positive</td>
</tr>
<tr>
<td>Period</td>
<td>Positive</td>
</tr>
<tr>
<td>Performance</td>
<td>Negative, but trivial</td>
</tr>
<tr>
<td>Availability</td>
<td>Same</td>
</tr>
<tr>
<td>Ease of use</td>
<td>Same</td>
</tr>
<tr>
<td>Cost</td>
<td>Same</td>
</tr>
<tr>
<td>Non-deterministic feature</td>
<td>Added</td>
</tr>
</tbody>
</table>
MaD1 - Statistical Test

(a) MaD1 - Statistical Test

(b) MaD1 - Statistical Test

<table>
<thead>
<tr>
<th>Battery</th>
<th>Parameters</th>
<th>NoP</th>
<th>d</th>
<th>nd</th>
<th>d-nd</th>
<th>nd-nd</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmallCrush</td>
<td>Built-in</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Crush</td>
<td>235 random numbers</td>
<td>144</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BigCrush</td>
<td>238 random numbers</td>
<td>160</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Rabbit</td>
<td>32x109 bits</td>
<td>40</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Alphabit</td>
<td>32x109 bits</td>
<td>17</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BlockAlphabit</td>
<td>32x109 bits</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
## MaD1 – Performance Test

### Pseudorandom Number Generation Speed (cycle/byte)

<table>
<thead>
<tr>
<th>Generator</th>
<th>Sequence size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>RC4</td>
<td>9.53</td>
</tr>
<tr>
<td>HC-128</td>
<td>55.21</td>
</tr>
<tr>
<td>MaD1 (32-bit)</td>
<td>47.97</td>
</tr>
<tr>
<td>MaD1 (64-bit)</td>
<td>38.70</td>
</tr>
</tbody>
</table>
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