MaD2: An Ultra-Performance Stream Cipher for Pervasive Data Encryption

Jie Li and Jianliang Zheng
The City University of New York

Contents

Motivation (1)

- Pervasive data encryption
  - Encrypt all data.
  - Encrypt data while they are moving on the wire and at rest as well.
  - Not all data are secrets and need to be secured, but such a need arises when
    - data classification is difficult or expensive (e.g., in cloud computing) or
    - encryption needs to be done at a layer below the application (e.g., for whole disk or communication channel encryption).

Motivation (2)

- Encryption at rest may affect the usability of some applications, but it is getting popular nowadays for several reasons:
  - Attacks keep growing in both quantity and complexity.
  - Virtual computing and cloud computing makes it difficult or impossible to put up a physical defense line against attacks.
  - The risk of losing sensitive data increases due to the wide use of portable computing devices.
  - Data at rest contain much more information than the data on the wire.

Motivation (3)

- Current popular ciphers
  - AES is too slow.
  - RC4 is about twice as fast as AES, but has some security issues.
  - Some eStream finalists are faster than RC4, but still slow down disk I/O by a factor of 1.5 or more.
- MaD2 is an ultra-performance stream cipher that can be used for pervasive data encryption.

Algorithm Details (1)

Notation

Notation	Usage
#	starting a comment line
++	increment (x++ is same as x = x + 1)
%	modulo
<<	left logical bitwise shift
>>	right logical bitwise shift
&	bitwise AND
|	bitwise OR
^	bitwise XOR
[ ]	array subscripting (subscript starts from 0)

Hexadecimal numbers are prefixed by "0x" and all variables and constants are unsigned integers in little endian.

Algorithm Details (2)

Key Scheduling

MaD2

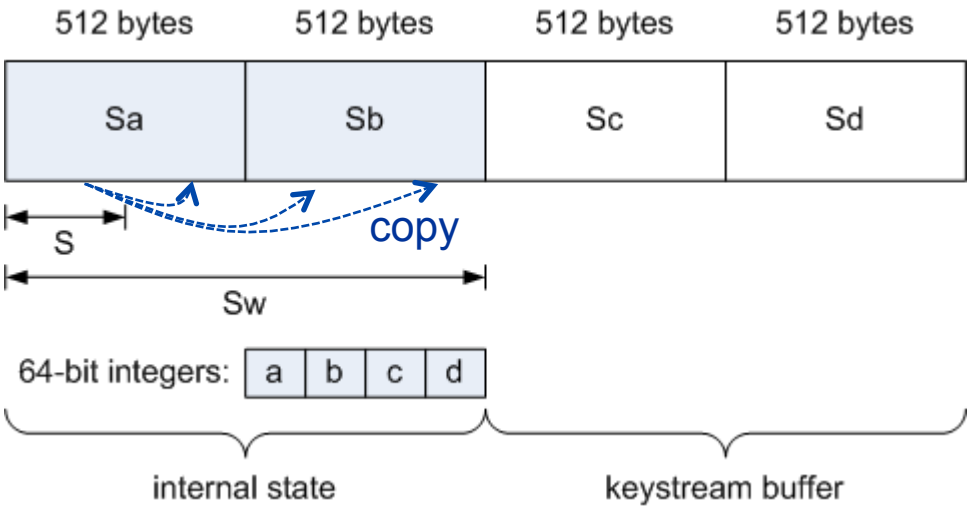
```
for i from 0 to 255
    S[i] = i
endfor
i = 0
j = 0
k = 0
for r from 0 to 319
    j = j + S[i] + key[i % szKey]
    k = k ^ j
    left_rotate (S[i], S[j],S[k])
    i++
endfor
i = j + k
```

RC4

```
for i from 0 to 255
    S[i] = i
endfor
j = 0
for i from 0 to 255
    j = j + S[i] + key[i % szKey]
    swap (S[i], S[j])
endfor
```

Algorithm Details (3)

State Initialization



Shuffle S (i.e., the first half of Sa) as follows:

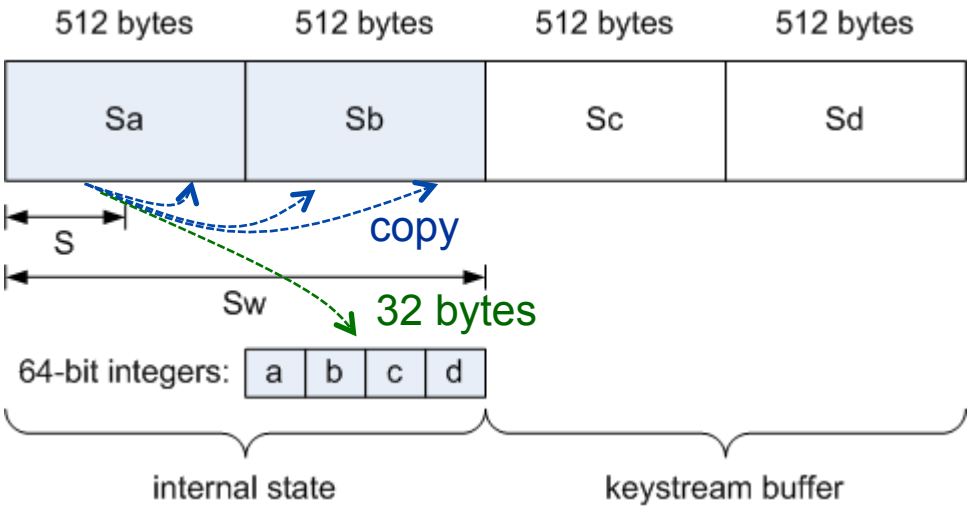
```
for r from 0 to 255
    i++
    j = j + S[i]
    k = k ^ j
    left_rotate (S[i], S[j], S[k])
endfor
```

Initialization steps:

1. Initialize *S* (i.e., the first half of *Sa*) through key scheduling.
2. Copy *S* to the second half of *Sa*.
3. Shuffle *S*.
4. Copy *S* to the first half of *Sb*.
5. Shuffle *S*.
6. Copy *S* to the second half of *Sb*.
7. Shuffle *S*.
8. Initialize *a*, *b*, *c*, and *d* (see next slide).

Algorithm Details (4)

State Initialization (cont.)



Generate 32 bytes from *S* and cast them into four 64-bit integers *a*, *b*, *c*, and *d* using little endian.

This completes the state initialization and now the internal state contains four permutations of {0, 1, ..., 255} and four initialized 64-bit integers.

```
# Generate 32 bytes from S.
for r from 0 to 7
    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]
endfor
```

Algorithm Details (5)

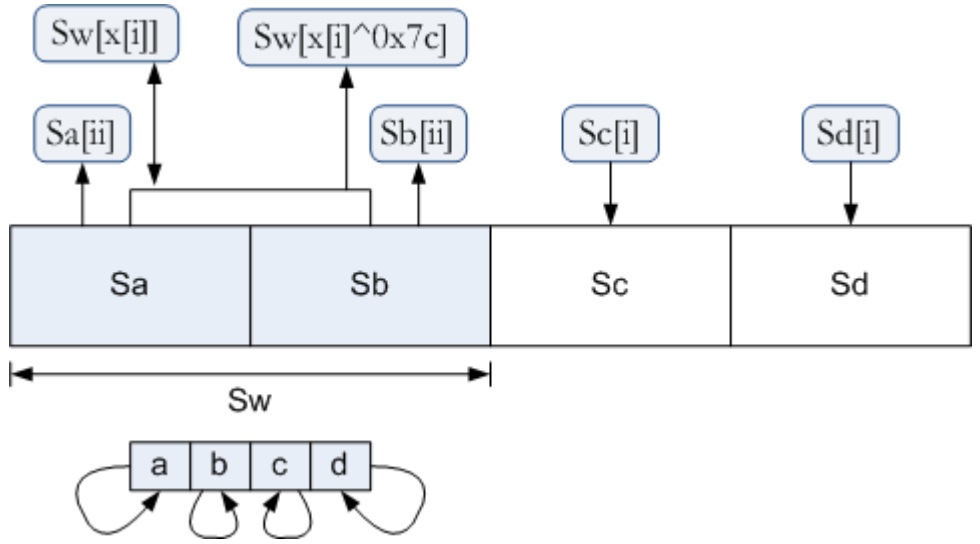
Keystream Generation

```
# declare a byte array of size 64 and
# cast it into 64-bit integer array
byte x[64] => x64[8]

# populate array x ( through x64)
m = 0x7c7c7c7c7c7c7c7c
n = 0x0302000103020001
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

# compute e, f, g, and h
e = a + Sw[a >> 57]
f = b + Sw[b >> 57]
```

```
g = c + Sw[c >> 57]
h = d + Sw[d >> 57]

# output and update the internal state
ii = 0
for i from 0 to 63
    a = a << 1
    b = b >> 1
    ii = ii ^ i
    a = a + (e ^ Sw[x[i]])
    b = b + (f ^ Sw[x[i]^0x7c])
    c = c + (g ^ Sa[ii])
    d = d + (h ^ Sb[ii])
    ii = ii & 1
    Sc[i] = c ^ (a + d)
    Sd[i] = d ^ (b + c)
    Sw[x[i]] = a + b
endfor
```

Algorithm Details (6)

Keystream Generation (cont.)



Data flow during one iteration

Algorithm Details (7)

Keystream Generation (cont.)

- Integer *ii* is computed in such a way that its lower two bits cycle through the values 0, 1, 3, 2 instead of the normal 0, 1, 2, 3, but otherwise it is same as the counter *i*.
- The four state table integers accessed during each iteration are distinct and they are also different from any of the four state table integers used in the previous or next iteration.

State table integer	Subscript	State table accessed	Subscript value (last 2 bits)
Sw[x[i]]	x[i]	Sy (= Sa or Sb)	3, 2, 0, 1, …
Sw[x[i]^0x7c]	x[i]^0x7c	Sz (= Sa or Sb, Sz ≠ Sy)	3, 2, 0, 1, …
Sa[ii]	ii	Sa	0, 1, 3, 2, …
Sb[ii]	ii	Sb	0, 1, 3, 2, …

Security Analysis

- Period length
  - Internal state size: 8448 bits
  - Period length: $\sim 2^{4224} \approx 3.55 \times 10^{1271}$
- Against known attacks
  - Attacks against RC4
    - Correlation attacks
    - Weak keys
    - Related key attacks
    - Linear approximations
  - Time-memory tradeoff attacks
  - Guessing attacks
  - Algebraic attacks
  - Distinguishing attacks

Statistical Testing

- NIST statistical test suite
  - 1000 sequences, each containing one million bits (125 KB)
  - examining the proportion of sequences that pass a statistical test and checking the distribution of *P-values* for uniformity
  - no failures
- Diehard battery of tests
  - setup
    - 100 sequences, each containing 96 million bits (12 MB)
    - 50 sequences, each containing 2176 million bits (272 MB)
  - checking the distribution of *P-values* for uniformity
  - no failures

Statistical Testing (cont.)

- Testu01 batteries of tests
  - 6 batteries
    - SmallCrush
    - Crush
    - BigCrush
    - Rabbit
    - Alphabit
    - BlockAlphabit
  - built-in parameters used for SmallCrush, Crush, and BigCrush
  - bit sequence size set to 32×10^9 for Rabbit, Alphabit, and BlockAlphabit
  - checking *P-values*
    - successful if a *P-value* falls in [0.001, 0.9990]
    - failed if a *P-value* is outside [10^{-10}, 1- 10^{-10}] (i.e., too close to 0 or 1)
    - in doubt otherwise
  - no failures

Performance Testing

- C implementation
- Microsoft Visual C/C++ Optimizing Compiler Version 16 with option /O2 (optimized for maximum speed)
- Intel Core i3 370M, 2.4GHz, 64 KB L1 data cache, 64 KB L1 instruction cache, 512 KB L2 cache
- Testing results (cycle/byte):

Generator	Keystream 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
MaD2 (32 bit)	51.45	12.16	7.83	3.46	2.99	2.98
MaD2 (64 bit)	42.08	9.05	5.07	1.30	0.93	0.91

Thank You!