Recent Topics on Symmetric Ciphers
- Security and implementation of S-box -

October 5 2006
Mitsuru Matsui
Mitsubishi Electric Corporation
Overview

• Trends of Block/Hash Primitives and Intel Processors

• Security Issues on S-box
  – Differential cryptanalysis: Security and related open problems
  – Linear cryptanalysis: Security and related open problems

• Implementation Issues on S-box
  – Processor Architecture of Pentium and Athlon
  – Ordinary Implementation of AES
  – Bitslice Implementation of AES and Camellia
Block/Hash primitives & Intel Processors

**RED:** lookup tables & logical
**BLUE:** arithmetic & logical

**1976:** DES (for hardware)

**1971:** 4004 (4bit,4KB,740KHz) First processor

**1974:** 8080 (8bit,64KB,2MHz)

**1978:** 8086 (16bit,1MB,5-10MHz) Segment

**1982:** 80286 (16bit,16MB,6-12.5MHz) Protect mode

**1985:** 80386 (32bit,4GB,16-33MHz) Virtual memory

**1989:** 80486 (25-100MHz) on chip L1 cache

**1993:** Pentium (60-200MHz) Superscalar

**1995:** Pentium Pro (150-200MHz)

**1997:** Pentium II (233-1300MHz) 64-bit MMX

**1999:** Pentium III (450-1400MHz) SSE

**2000:** Pentium 4 (-3.4GHz) SSE2 “Northwood”

**2003:** Pentium M (-2.1GHz)

**2004:** Pentium 4 (-3.8GHz) SSE3 “Prescott” EM64T

**2006:** Core (-2.33GHz)

**2006:** Core2(-2.93GHz) SSE4 EM64T

**1987:** RC2 (16bit), FEAL (8bit)

**1989:** MD2 (16bit)

**1990:** MD4 (32bit), Multi2 (32bit)

**1991:** IDEA (16bit)

**1992:** MD5 (32bit)

**1994:** RC5 (32bit)

**1995:** SHA-1 (32bit)

**1996:** MISTY1

**1998:** AES, RC6, Serpent, Mars, Blowfish

**2000:** Kasumi, Camellia, Whirlpool (64bit)

**2002:** SHA-2 (32,64bit)

**2004:** ARIA

**2006:** SHA-2 (32,64bit)
S-Box - a lookup table -

• **6-in/4-out**
  – DES  design criteria unknown

• **7-in/7-out, 9-in/9-out**
  – MISTY  a power function over a Galois field

• **8-in/8-out**
  – AES, Camellia, ARIA (block ciphers)
  – SNOW, MUGI (stream ciphers)

  an inversion over Galois field $GF(2^8)$
Why an inversion over $GF(2^8)$?

(+)  
- Suitable for software implementation  
- Believed (but not proved) to be strongest against differential and linear cryptanalysis

(-)  
- Might be weak against algebraic attacks
Differential attacks and S-box

**Differential Uniformity:**

\[ DP_S(dx,dy) \overset{\text{def}}{=} \#\{x\mid S(x+dx)+S(x)=dy\} \]

**Strength against differential attacks:**

\[ DP_S \overset{\text{def}}{=} \max_{dx \neq 0, dy} DP_S(dx,dy) \]

(1) \( DP_S \geq 2 \) for any \( S \).

(2) If \( S(x)=x^3 \) then \( DP_S = 2 \) for odd \( n \).

(3) If \( S(x)=1/x \) (\( S(0)=0 \)) then \( DP_S = 4 \) for even \( n \).
Open Problems 1

(I) Find a bijective function $S$ over $\text{GF}(2^{2m})$ such that $\text{DP}_S = 2$.

(II) Find a bijective function $S$ over $\text{GF}(2^{2m})$ such that $\text{DP}_S = 4$ and $S$ is not linearly equivalent to an inversion.

Remark:
Probably (I) does not exist. Confirmed for $m=2$. 
Linear attacks and S-box

Nonlinearity:

\[ LP_S(m_x, m_y) \overset{\text{def}}{=} |\#\{x|m_x \cdot x = m_y \cdot S(x)\} - 2^{n-1}| \]

\[ \text{Strength against linear attacks:} \]

\[ LP_S \overset{\text{def}}{=} \max_{m_x, m_y \neq 0} LP_S(m_x, m_y) \]

(1) \( LP_S \geq 2^{(n-1)/2} \) for any \( S \).
(2) If \( S(x) = x^3 \), then \( LP_S = 2^{(n-1)/2} \) for odd \( n \).
(3) If \( S(x) = 1/x \) (\( S(0) = 0 \)), then \( LP_S \geq 2^{n/2} \) for even \( n \).
Open Problems 2

(I) Find a function $S$ over $\mathbb{GF}(2^{2m})$ such that $2^{(2m-1)/2} < LPS < 2^m$.

(II) Find a bijective function $S$ over $\mathbb{GF}(2^8)$ not linearly equivalent to an inversion such that $LPS = 2^4$.

Remark:

Probably (I) does not exist. Confirmed for $m=2$. 
# x86 Architecture

<table>
<thead>
<tr>
<th>32bit</th>
<th>64bit</th>
<th>128bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>mm0</td>
<td>xmm0</td>
</tr>
<tr>
<td>ebx</td>
<td>mm1</td>
<td>xmm1</td>
</tr>
<tr>
<td>ecx</td>
<td>mm2</td>
<td>xmm2</td>
</tr>
<tr>
<td>edx</td>
<td>mm3</td>
<td>xmm3</td>
</tr>
<tr>
<td>esi</td>
<td>mm4</td>
<td>xmm4</td>
</tr>
<tr>
<td>edi</td>
<td>mm5</td>
<td>xmm5</td>
</tr>
<tr>
<td>ebp</td>
<td>mm6</td>
<td>xmm6</td>
</tr>
<tr>
<td>esp</td>
<td>mm7</td>
<td>xmm7</td>
</tr>
</tbody>
</table>

**CISC Instruction Set**

- `xor  eax, [esi+ebx]`
- `add  12[ebp], al`

**Source and Destination**

- Destination
- Source
Encryption speed of Gladman’s Serpent assembly codes optimized for P3

<table>
<thead>
<tr>
<th></th>
<th>Pentium III Coppermine</th>
<th>Pentium 4 Northwood</th>
<th>Pentium 4 Prescott</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit x86:</td>
<td>Straightforward</td>
<td>773</td>
<td>1267</td>
</tr>
<tr>
<td>64-bit MMX:</td>
<td>2-block parallel</td>
<td>570</td>
<td>1052</td>
</tr>
<tr>
<td>128-bit XMM:</td>
<td>4-block parallel</td>
<td>-</td>
<td>656</td>
</tr>
</tbody>
</table>

- **32-bit x86**
  - block 1

- **64-bit MMX**
  - block 1
  - block 2

- **128-bit XMM**
  - block 1
  - block 2
  - block 3
  - block 4
Micro-operations (μops)

- Pentium instructions are decomposed into RISC-style simple operations (μops) at the decoding stage
  - Intel has not published exact details on μops
- Programmers cannot directly read/write a code of micro-operations

```
xor eax,[mem]  
```

A Pentium instruction

```
{ load reg1,[mem]  
xor reg2,reg1  
}
```

Corresponding μops
How to measure performance

xor eax,eax
cpuid
rdtsc
mov CLK1,eax
xor eax,eax
cpuid

Encryption(...,block) /* nothing */

xor eax,eax
cpuid
rdtsc
mov CLK2,eax
xor eax,eax
cpuid

( (CLK2-CLK1) – (CLK4-CLK3) ) / block
Difficulties in Measurement

- **Common Implicit Assumptions**
  - Should run in a constant time without interruptions
  - Should take more cycles if an interruption takes place

- **These assumptions do not hold on Pentium 4 (?)**

<table>
<thead>
<tr>
<th>HT: Hyperthread</th>
<th>Northwood w/o HT</th>
<th>Northwood with HT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most frequent cycles</td>
<td>632 cycles</td>
<td>636 cycles</td>
</tr>
<tr>
<td>Minimum cycles</td>
<td>632 cycles</td>
<td>600 cycles (very rare)</td>
</tr>
</tbody>
</table>

“Overhead” measurement results

Also Prescott Stepping 3 Revision 0 looks unstable
Advanced Encryption Standard

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
  byte state[4,Nb]

  state = in

  AddRoundKey(state, w[0, Nb-1])  // See Sec. 5.1.4

  for round = 1 step 1 to Nr-1
    SubBytes(state)  // See Sec. 5.1.1
    ShiftRows(state)  // See Sec. 5.1.2
    MixColumns(state) // See Sec. 5.1.3
    AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
  end for

  SubBytes(state)
  ShiftRows(state)
  AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

  out = state
end
```

Figure 5. Pseudo Code for the Cipher.¹
Figure 6. SubBytes() applies the S-box to each byte of the State.

\[
\begin{bmatrix}
S_{0,c} \\
S_{1,c} \\
S_{2,c} \\
S_{3,c}
\end{bmatrix}
= \begin{bmatrix}
02 & 03 & 01 & 01 \\
01 & 02 & 03 & 01 \\
01 & 01 & 02 & 03 \\
03 & 01 & 01 & 02
\end{bmatrix}
\begin{bmatrix}
S_{0,c} \\
S_{1,c} \\
S_{2,c} \\
S_{3,c}
\end{bmatrix}
\]

Figure 7. ShiftRows() cyclically shifts the last three rows in the State.

Figure 8. ShiftRows() cyclically shifts the last three rows in the State.

Figure 9. MixColumns() operates on the State column-by-column.

Figure 10. AddRoundKey() XORs each column of the State with a word from the key schedule.
One round of AES is simple

**ShiftRow+SubBytes+MixColumn**

\[
\begin{align*}
A' &= T0[A_0] \oplus T1[B_1] \oplus T2[C_2] \oplus T3[D_3] \\
B' &= T0[B_0] \oplus T1[C_1] \oplus T2[D_2] \oplus T3[A_3] \\
C' &= T0[C_0] \oplus T1[D_1] \oplus T2[A_2] \oplus T3[B_3] \\
D' &= T0[D_0] \oplus T1[A_1] \oplus T2[B_2] \oplus T3[C_3]
\end{align*}
\]

**AddRoundKey**

\[
\begin{align*}
A &= A' \oplus \text{KeyA} \\
B &= B' \oplus \text{KeyB} \\
C &= C' \oplus \text{KeyC} \\
D &= D' \oplus \text{KeyD}
\end{align*}
\]

A, B, C, D, A', B', C', D': 4-byte data

A_i: i-th byte of A

Ti: 1KB table (1byte->4bytes)

Another tables in the final round
AES round function in x86

ShiftRow+SubBytes+MixColumn can be done by a four-time repetition of the following sequence:

```
movzx     esi, cl
mov/xor   reg32_2, T2[esi*4]
movzx     esi, ch
mov/xor   reg32_1, T1[esi*4]
shr       ecx, 16
movzx     esi, cl
mov/xor   reg32_0, T0[esi*4]
movzx     esi, ch
mov/xor   reg32_3, T3[esi*4]
```
Our implementation of AES

<table>
<thead>
<tr>
<th></th>
<th>Pentium III</th>
<th>Pentium 4 Northwood</th>
<th>Pentium 4 Prescott</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mu \text{ops} / \text{block} )</td>
<td>596</td>
<td>654</td>
<td>654</td>
</tr>
<tr>
<td>\text{cycles} / \text{block}</td>
<td>232</td>
<td>251</td>
<td>284</td>
</tr>
<tr>
<td>\text{cycles} / \text{byte}</td>
<td>14.5</td>
<td>15.7</td>
<td>17.8</td>
</tr>
<tr>
<td>( \mu \text{ops} / \text{cycles} )</td>
<td>2.57</td>
<td>2.61</td>
<td>2.30</td>
</tr>
</tbody>
</table>

Slow in Prescott probably due to its high load latency
x86 vs. x64: Registers

<table>
<thead>
<tr>
<th>64bit/32bit</th>
<th>128bit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>rax</strong></td>
<td>eax</td>
</tr>
<tr>
<td></td>
<td>ebx</td>
</tr>
<tr>
<td></td>
<td>ecx</td>
</tr>
<tr>
<td></td>
<td>edx</td>
</tr>
<tr>
<td></td>
<td>esi</td>
</tr>
<tr>
<td></td>
<td>edi</td>
</tr>
<tr>
<td></td>
<td>ebp</td>
</tr>
<tr>
<td></td>
<td>esp</td>
</tr>
<tr>
<td><strong>r8</strong></td>
<td>r8d</td>
</tr>
<tr>
<td><strong>r9</strong></td>
<td>r9d</td>
</tr>
<tr>
<td><strong>r10</strong></td>
<td>r10d</td>
</tr>
<tr>
<td><strong>r11</strong></td>
<td>r11d</td>
</tr>
<tr>
<td><strong>r12</strong></td>
<td>r12d</td>
</tr>
<tr>
<td><strong>r13</strong></td>
<td>r13d</td>
</tr>
<tr>
<td><strong>r14</strong></td>
<td>r14d</td>
</tr>
<tr>
<td><strong>r15</strong></td>
<td>r15d</td>
</tr>
</tbody>
</table>
x64: Better and Worse

(+) more registers, longer registers
(+) most instructions have a 64-bit form
   ex) rol reg32,8 => rol reg64,8

(-) longer instruction, inefficient decoding
   a prefix byte needed for an extended instruction form.
(-) a 64-bit instruction is not always fast
   ex) “shift” and “rotate” on Pentium 4
Pentium 4 vs. Athlon 64

**Pentium 4 (Prescott core)**  up to 3.8GHz
(+) long pipeline stages, high clock frequency
(+) instructions are cached after being decoded
(−) poorly documented, never works as Intel claims

**Athlon 64**  up to 2.8GHz
(+) high superscalability (5 uops/cycle)
(+ ) well documented, less frustrating for programmers
(−) its decoding stage can be a bottleneck
## Instruction Latency/Throughput

<table>
<thead>
<tr>
<th>Processor</th>
<th>Pentium 4 Prescott (EM64T)</th>
<th>Athlon 64 (AMD64)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operand Size</td>
<td>32-bit</td>
<td>64-bit</td>
</tr>
<tr>
<td>mov reg, [mem]</td>
<td>4, 1</td>
<td>4, 1</td>
</tr>
<tr>
<td>mov reg, reg</td>
<td>1, 2.88</td>
<td>1, 2.88</td>
</tr>
<tr>
<td>add/sub reg, reg</td>
<td>1, 2.88</td>
<td>1, 2.88</td>
</tr>
<tr>
<td>xor/and/or reg, reg</td>
<td>1, 2</td>
<td>1, 2</td>
</tr>
<tr>
<td>shr reg, imm</td>
<td>1, 1.75</td>
<td>7, 1</td>
</tr>
<tr>
<td>shl reg, imm</td>
<td>1, 1.75</td>
<td>1, 1.75</td>
</tr>
<tr>
<td>ror/rol reg, imm</td>
<td>1, 1</td>
<td>7, 0.14-1</td>
</tr>
</tbody>
</table>

To slow 64-bit right shifts and 64-bit rotations
Rotate shifts on 64-bit Pentium 4

```
rol rax,1
rol rbx,1
rol rcx,1
rol rdx,1
rol rsi,1
rol rdi,1
rol rbp,1

xor r9, r9
xor r9, r9
xor r9, r9
xor r9, r9
xor r9, r9
xor r9, r9

49 cycles (throughput : 1/7)  7 cycles (throughput : 1)
```
### Some Code Examples

<table>
<thead>
<tr>
<th></th>
<th>32 bit</th>
<th>64 bit (1)</th>
<th>64 bit (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>xor eax,0[esi+ecx]</code></td>
<td><code>xor rax,0[rsi+rcx]</code></td>
<td><code>xor rax,TABLE+0[rcx]</code></td>
</tr>
<tr>
<td></td>
<td><code>xor ebx,4[esi+ecx]</code></td>
<td><code>xor rbx,8[rsi+rcx]</code></td>
<td><code>xor rbx,TABLE+8[rcx]</code></td>
</tr>
<tr>
<td></td>
<td><code>add ecx,8</code></td>
<td><code>add rcx,16</code></td>
<td><code>add rcx,16</code></td>
</tr>
<tr>
<td>Length</td>
<td>10 bytes</td>
<td>13 bytes</td>
<td>18 bytes</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2.2 cycles</td>
<td>2.2 cycles</td>
<td>2.2 cycles</td>
</tr>
<tr>
<td>Athlon 64</td>
<td>1.0 cycle</td>
<td>1.0 cycle</td>
<td>1.4 – 1.9 cycles</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>32 bit</th>
<th>64 bit (1)</th>
<th>64 bit (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>movzx ecx,al</code></td>
<td><code>movzx rcx,al</code></td>
<td><code>movzx rcx,al</code></td>
</tr>
<tr>
<td></td>
<td><code>xor ebx,[esi+ecx*4]</code></td>
<td><code>xor rbx,[rsi+rcx*8]</code></td>
<td><code>xor rbx,TABLE[rcx*8]</code></td>
</tr>
<tr>
<td></td>
<td><code>shr eax,8</code></td>
<td><code>shr rax,8</code></td>
<td><code>shr rax,8</code></td>
</tr>
<tr>
<td>Length</td>
<td>9 bytes</td>
<td>12 bytes</td>
<td>16 bytes</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>1.7 cycles</td>
<td>7.0 cycles</td>
<td>7.0 cycles</td>
</tr>
<tr>
<td>Athlon 64</td>
<td>1.0 cycle</td>
<td>1.0 cycle</td>
<td>1.0 cycle</td>
</tr>
</tbody>
</table>
Performance of AES on x64 Processors

- The structure of AES is optimized for 32-bit processors.
- Free from “register starvation” due to 16 general registers.

Performance of AES (128-bit key) on Athlon64/Pentium 4

<table>
<thead>
<tr>
<th>Processors</th>
<th>AES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Athlon 64</td>
</tr>
<tr>
<td></td>
<td>64-bit</td>
</tr>
<tr>
<td>cycles/block</td>
<td>170</td>
</tr>
<tr>
<td>instructions/cycle</td>
<td>2.74</td>
</tr>
<tr>
<td>uops/cycle</td>
<td>3.53</td>
</tr>
</tbody>
</table>
Bitslice Implementation of Block Ciphers

- Introduced by Biham (FSE’97)
- \( n \)-block parallel execution using \( n \)-bit registers
- 1 software instruction = \( n \) simple hardware gates
  - AND, OR, XOR, NOT…
- Very efficient if
  - registers are long
  - registers are many
  - the target algorithm is small in hardware
- Protection against cache timing attack
Principle of Bitslice Implementation

Ex) \texttt{xor reg1,reg2}

is an \textit{n}-parallel execution of 2-bit-input/1-bit-output XOR of each block.
Bitslice and S-box

- Many recent block ciphers have adopted an 8x8 S-box (a lookup table), linearly equivalent to an inversion over GF(2^8).
  - AES, Camellia, SNOW2.0, ARIA etc

- An inversion over GF(2^8) is strong against differential/linear attacks (actually best known), but can be weak against cache timing attacks.

- The bitslice implementation can compute an inversion over GF(2^8) without a table lookup.
Multiplication over \( \text{GF}(2^{2n}) \) using \( \text{GF}(2^n) \)

Basis of \( \text{GF}(2^{2n})/\text{GF}(2^n) \): \((1, a)\)

\[
Z_0 + Z_1 a = (X_0 + X_1 a)(Y_0 + Y_1 a) \quad \text{where} \quad \text{Tr}(a) = 1
\]

3 multiplications over \( \text{GF}(2^n) \)
Inversion over $\text{GF}(2^{2n})$ using $\text{GF}(2^n)$

\[ Z_0 + Z_1 a = \frac{1}{X_0 + X_1 a} \quad \text{where} \quad \text{Tr}(a) = 1 \]
Circuits on $\text{GF}(2^2)$

**Addition**
- XOR $x_0, x_1$
- XOR $y_0, y_1$

**Multiplication**
1. MOV $t_0, x_1$ ; $t_0$ temporary
2. XOR $x_1, x_0$
3. AND $x_0, y_0$
4. AND $t_0, y_1$
5. XOR $y_0, y_1$
6. AND $x_1, y_0$
7. XOR $x_1, x_0$
8. XOR $x_0, t_0$ ; $y_1$ unchanged

**Inversion**
- AND $x_0, x_1$
- NOT $x_0$
Multiplication/Inversion on GF(2^4)

**Multiplication**
- 3 reg copies
- 35 instructions with 3 temp. registers

**Inversion**
- 4 reg copies
- 34 instructions with 4 temp. registers
Inversion on GF($2^8$)

177 instructions (156 reg-reg’s + 21 mem-reg’s)
Implementation Results

The Full AES S-box

<table>
<thead>
<tr>
<th>Basis Change (before inversion)</th>
<th>Inversion</th>
<th>Basis Change (after inversion)</th>
<th>Constant XOR</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>177</td>
<td>16</td>
<td>(4)</td>
<td>205 (209)</td>
</tr>
</tbody>
</table>

Performance of Bitsliced AES/Camellia on Athlon64/Pentium 4

<table>
<thead>
<tr>
<th></th>
<th>AES</th>
<th>Camellia</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processors</td>
<td>Athlon 64</td>
<td>Pentium 4</td>
</tr>
<tr>
<td>cycles/block</td>
<td>250</td>
<td>418</td>
</tr>
<tr>
<td>instructions/cycle</td>
<td>2.75</td>
<td>1.66</td>
</tr>
<tr>
<td>uops/cycle</td>
<td>3.20</td>
<td>1.93</td>
</tr>
</tbody>
</table>
Concluding Remarks

• A combination of lookup tables and logical operations is suitable for both software and hardware.

• Understanding hardware is important in doing software.

• Pentium 4 looks a dead end of processor design
  – The long pipeline leads to an overheating problem
  – AMD Athlon64 very often runs faster than Pentium 4

• Parallel encryption will be increasingly important

• Intel’s new ‘Core’ processors go back to Pentium III
  – Bitsliced ciphers can be much faster on Core2