EXPERIENCE IN AES ALGORITHM IMPLEMENTATION

• Byte Order Problems in the AES Specifications
• Implementation and Optimisation in C for the Pentium II processor
• Serpent Optimisation
• Performance Results
• My AES Winners and Losers
Is E2 'Big-Endian' or 'Little-Endian'?

3. For an element \((a_{n-1}, a_{n-2}, \ldots, a_0)\) of set \(A^n\), let \(a_{n-1}\) be the left most element, and \(a_0\) be the right most element.

12. An element \((a_7, a_6, \ldots, a_0)\) in the set \(B\), where \(a_i \in GF(2)\), is identified with

\[
\sum_{i=0}^{7} a_i 2^i \mod 2^8 \mathbb{Z} \in \mathbb{Z}/2^8\mathbb{Z},
\]

where \(a_i \in GF(2)\) \((i = 0, 1, \ldots, 7)\) corresponds to \(\hat{a}_i \in \{0, 1\} \subset \mathbb{Z}\) in a canonical way, i.e., \(a_7\) is the most significant (left most) bit and \(a_0\) is the least significant (right most) bit.

13. An element \((b_3, b_2, b_1, b_0)\) in the set \(W\), where \(b_i \in B\), is identified with

\[
\sum_{i=0}^{3} b_i 2^{8i} \mod 2^{32} \mathbb{Z} \in \mathbb{Z}/2^{32}\mathbb{Z},
\]

where \(b_i \in B\) \((i = 0, 1, 2, 3)\) corresponds to \(\hat{b}_i \in \{0, 1, \ldots, 2^8 - 1\} \subset \mathbb{Z}\). The correspondence of \(b_i\) to \(\hat{b}_i\) is defined in item 12.
HPC and Serpent I/O Byte Order?

**HPC:** Bits are numbered from left to right, with bit 63 being the leftmost bit of a word, and also the numerically largest.

If an ascii character string is used as a key, the characters are placed into an array of 64-bit words, right-to-left. The first character of the string will occupy bit positions 7-0, the second character will occupy bit positions 15-8, etc. Within a character,

When hexadecimal data is presented to Hasty Pudding, a different convention is used: Complete words are filled in from left to right,

**Serpent:** streams. The indices of the bits are counted from 0 to bit 31 in one 32-bit word, 0 to bit 127 in 128-bit blocks, 0 to bit 255 in 256-bit keys, and so on. For internal computation, all values are represented in little-endian, where the first word (word 0) is the least significant word, and the last word is the most significant, and where bit 0 is the least significant bit of word 0. Externally, we write each block as a plain 128-bit hex number.

- External to what? The Cipher? The Program?
- And what exactly is a ‘plain’ 128-bit hex number?
- Non-portable if written in the machine format
## Specifications - Byte Order

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Specified I/O byte order</th>
<th>Internal byte order</th>
<th>Required action on a little-endian processor to match supplied test vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>RC6</td>
<td>little-endian</td>
<td>neutral</td>
<td>none</td>
</tr>
<tr>
<td>Rijndael</td>
<td>implied little-endian</td>
<td>neutral</td>
<td>none</td>
</tr>
<tr>
<td>MARS</td>
<td>little-endian</td>
<td>neutral</td>
<td>none</td>
</tr>
<tr>
<td>TWOFISH</td>
<td>little-endian</td>
<td>little-endian</td>
<td>none</td>
</tr>
<tr>
<td>CRYPTON</td>
<td>little-endian</td>
<td>little-endian</td>
<td>none</td>
</tr>
<tr>
<td>CAST-256</td>
<td>none</td>
<td>neutral</td>
<td>invert byte order in each 32-bit word</td>
</tr>
<tr>
<td>E2</td>
<td>!</td>
<td>!</td>
<td>invert byte order in each 32-bit word</td>
</tr>
<tr>
<td>Serpent</td>
<td>?</td>
<td>little-endian</td>
<td>invert byte order of 16 byte block</td>
</tr>
<tr>
<td>HPC</td>
<td>? (64 bit)</td>
<td>neutral</td>
<td>invert byte order in each 64-bit word</td>
</tr>
<tr>
<td>DFC</td>
<td>big-endian</td>
<td>big-endian</td>
<td>invert byte order in each 32-bit word</td>
</tr>
<tr>
<td>SAFER+</td>
<td>big-endian</td>
<td>neutral</td>
<td>invert byte order of 16 byte block</td>
</tr>
<tr>
<td>LOKI97</td>
<td>? (64 bit)</td>
<td>neutral</td>
<td>invert byte order in each 32-bit word</td>
</tr>
<tr>
<td>FROG</td>
<td>little-endian</td>
<td>neutral</td>
<td>none</td>
</tr>
<tr>
<td>DEAL</td>
<td>little-endian</td>
<td>little-endian</td>
<td>none</td>
</tr>
<tr>
<td>MAGENTA</td>
<td>implied little-endian</td>
<td>little-endian</td>
<td>none</td>
</tr>
</tbody>
</table>
Pentium II Paranoia - RC6

The Pentium II has an apoplectic fit when asked to do a division

```
for(k = 0; k < 132; ++k)
{   a = rotl(l_key[i] + a + b, 3);
    b += a;
    b = rotl(l[j] + b, b);
    l_key[i] = a; l[j] = b;
    i = (i + 1) % 44;
    j = (j + 1) % t;
}
```

```
for(k = 0, t--; k < 132; ++k)
{   a = rotl(l_key[i] + a + b, 3);
    b += a;
    b = rotl(l[j] + b, b);
    l_key[i] = a; l[j] = b;
    i = (i == 43 ? 0 : i + 1);
    j = (j == t ? 0 : j + 1);
}
```

10364 cycles 1632 cycles

• Its not that the division operation is that bad
• But only one of the two parallel pipelines can do it
• So instruction scheduling gets rotted up
Pentium II - Register Renaming

• The left hand code sequence looks as if it should be faster since it only involves a single instruction, loading 1 byte

• That on the right loads 4 bytes and also has to perform a 32 bit shift operation

• But the right hand code is much faster because:
  • The PII can rename its visible registers using 40 invisible ones
  • The code on the right allows renaming because the new register value is unrelated to its previous value
  • The left hand code doesn’t because the top three bytes of the old register value are still being used
  • Hence the code on the left often stalls one or both pipelines
Pentium II - The Data Cache - Serpent

- Organised in 32 byte blocks - eight 32-bit words each
- One access in a block gets all 32 bytes into the cache
- Access to the other data items then comes almost ‘free’

```c
#define RND01(a,b,c,d,w,x,y,z)
{  register unsigned long t02, t03,
    t04, t05, t06, t07, t08,
    t10, t11, t12, t13, t16,
    t17, t01;
    t01 = a   | d  ; t02 = c   ^ d  ;
    t03 =     ~ b  ; t04 = a   ^ c  ; t05 = a   | t03; t06 = d   & t04; t07 = t01 & t02; t08 = b   | t06; y   = t02 ^ t05; ...
}
```

```c
#define sb1(a,b,c,d,e,f,g,h)
{  t1 = ~a       ; t2 = b ^ t1;
    t3 = a | t2   ; t4 = d | t2;
    t5 = c ^ t3   ; g = d ^ t5;
    t7 = b ^ t4   ; t8 = t2 ^ g;
    t9 = t5 & t7   ; h = t8 ^ t9;
    t11 = t5 ^ t7 ; f = h ^ t11;
    t13 = t8 & t11; e = t5 ^ t13
}
```

The Serpent encryption routine uses eight S boxes such as the one shown here

- With many C compilers the left hand code pulls in two cache blocks for EACH of the eight S boxes - 16 cache read/writes
- With the code on the right the whole encryption routine uses only two cache blocks
- This can improve Serpent speed by 10% or more
The Serpent S Box Boolean Functions - I

- Boolean functions with 4 input bits (coding 0-15) and 4 output bits (again coding 0-15), e.g:

<table>
<thead>
<tr>
<th>Input</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>15</td>
<td>12</td>
<td>2</td>
<td>7</td>
<td>9</td>
<td>0</td>
<td>5</td>
<td>10</td>
<td>1</td>
<td>11</td>
<td>14</td>
<td>8</td>
<td>6</td>
<td>13</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

- We want a circuit with AND, OR and NOT gates which gives the specified output states for each of the specified input states:

  \[
  t_0 = b \text{ xor} (\text{not} \ a) \\
  t_1 = c \text{ xor} (a \text{ or} t_0) \\
  g = d \text{ xor} t_1 \\
  t_2 = b \text{ xor} (d \text{ or} t_0) \\
  t_3 = t_0 \text{ xor} g \\
  h = t_3 \text{ xor} (t_1 \text{ and} t_2) \\
  t_4 = t_1 \text{ xor} t_2 \\
  f = h \text{ xor} t_4 \\
  e = t_1 \text{ xor} (t_3 \text{ and} t_4)
  \]

- We want the ‘minimum cost’ circuit - the one with the fewest Boolean operations
The Serpent S Box Boolean Functions - II

- Start with an initial list of 5 ‘primitives’
- Use a recursive function that:
  - adds a binary term that is a combination of existing terms using AND, OR or XOR
  - for all combinations of existing terms and for each operator
  - checking if e, f, g or h have been matched
  - if a match use this as a basis for a deeper recursion to match the remaining outputs
  - if no match add a recursion level

- This worked but it was painfully slow in finding improved S boxes
- After running it for several days on a 200 MHz PII, I had a couple of better S boxes
The Serpent S Box Boolean Functions - III

- Rather than checking if e, f, g or h have been matched,
- Check if the new term will combine with an existing list item to match e, f, g or h
- Pretty stupid since this involves a lot more work in the core of the recursive function!
- BUT it saves a level of recursion and pays off handsomely!
- Use limited processor power by preferring depth first recursion - build on existing partial solutions rather than looking for new partials

- I get good results by running my PII over a weekend, reducing the average S box function by about 1.5 Boolean terms
- This gets Serpent to 25 megabits/second on the PII reference platform
- All then goes quiet for a couple of months
The Serpent S Box Boolean Functions - IV

- Ross mentions in passing on the ‘UKCRYPTO’ mailing list that I have improved Serpent’s performance
- Several people email to ask how I did this. This includes Sam Simpson (of SCRAMDISK fame)
- Sam offers to run my program on some high capacity servers that he has access to and which lie mostly dormant at night and at weekends
- He tries and fails (at this stage there is no way any sane person can drive my program)
- I improve my program and convert it to run a width first search (not expecting any results because of the search depth this will need)
- Over about a week just before Christmas we get many new S box functions including two with only 14 terms.
- The new functions get Serpent to nearly 27 megabits/second
- So a combination of cache and Boolean function optimisations have improved Serpent speed by around 15%
## AES Candidate Performance - I

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Key Setup (128)</th>
<th>Encryption speed (128)</th>
<th>Decryption speed (128)</th>
<th>Mean speed (128)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RC6</td>
<td>1632</td>
<td>94.8</td>
<td>113.3</td>
<td>103.2</td>
</tr>
<tr>
<td>Rijndael</td>
<td>305:1389</td>
<td>68.4</td>
<td>72.7</td>
<td>70.2</td>
</tr>
<tr>
<td>MARS</td>
<td>4316</td>
<td>69.4</td>
<td>68.1</td>
<td>68.7</td>
</tr>
<tr>
<td>Twofish</td>
<td>9376</td>
<td>67.5</td>
<td>66.5</td>
<td>67.0</td>
</tr>
<tr>
<td>CRYPTON</td>
<td>531:1369</td>
<td>54.1</td>
<td>54.1</td>
<td>54.1</td>
</tr>
<tr>
<td>CRYPTON v1</td>
<td>744:1270</td>
<td>53.8</td>
<td>54.5</td>
<td>54.1</td>
</tr>
<tr>
<td>CAST</td>
<td>4333</td>
<td>40.4</td>
<td>40.4</td>
<td>40.4</td>
</tr>
<tr>
<td>E2</td>
<td>9473</td>
<td>37.3</td>
<td>37.0</td>
<td>37.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Key Setup (128)</th>
<th>Encryption speed (128)</th>
<th>Decryption speed (128)</th>
<th>Mean speed (128)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serpent</td>
<td>2402</td>
<td>26.9</td>
<td>28.0</td>
<td>27.4</td>
</tr>
<tr>
<td>HPC</td>
<td>120749</td>
<td>17.9</td>
<td>16.0</td>
<td>16.9</td>
</tr>
<tr>
<td>DFC</td>
<td>7166</td>
<td>15.6</td>
<td>15.4</td>
<td>15.5</td>
</tr>
<tr>
<td>SAFER</td>
<td>4278</td>
<td>14.9</td>
<td>15.0</td>
<td>14.9</td>
</tr>
<tr>
<td>LOKI</td>
<td>7430</td>
<td>12.0</td>
<td>11.7</td>
<td>11.8</td>
</tr>
<tr>
<td>FROG</td>
<td>1416182</td>
<td>10.6</td>
<td>11.5</td>
<td>11.0</td>
</tr>
<tr>
<td>DEAL</td>
<td>8635</td>
<td>10.9</td>
<td>10.8</td>
<td>10.9</td>
</tr>
<tr>
<td>MAGENTA</td>
<td>30</td>
<td>3.9</td>
<td>3.9</td>
<td>3.9</td>
</tr>
</tbody>
</table>

- Values are for the 200 MHz PII Reference Platform
- The compiler is Microsoft VC++ used in a pragmatic way
- Sensible non-ANSI optimisations (e.g. rotates) have been used
- Byte order inversion costs are not included
- Key set-up is in cycles, encryption/decryption in megabits/second
- Consistent code style, using no (overly) obscure techniques
## AES Candidate Performance - II

- Ranking of AES candidates for encrypting 1 block (16 bytes)

<table>
<thead>
<tr>
<th>Rijndael</th>
<th>CRYPTON</th>
<th>RC6</th>
<th>Serpent</th>
<th>MARS</th>
<th>CAST</th>
<th>SAFER</th>
<th>MAGENTA</th>
</tr>
</thead>
</table>

- Ranking of AES candidates for encrypting 256 blocks (4096 bytes)

<table>
<thead>
<tr>
<th>RC6</th>
<th>Rijndael</th>
<th>MARS</th>
<th>Twofish</th>
<th>CRYPTON</th>
<th>CAST</th>
<th>E2</th>
<th>Serpent</th>
</tr>
</thead>
</table>

- Ranking of AES candidates for bulk encryption (> 100000 bytes)

<table>
<thead>
<tr>
<th>RC6</th>
<th>MARS</th>
<th>Rijndael</th>
<th>Twofish</th>
<th>CRYPTON</th>
<th>CAST</th>
<th>E2</th>
<th>Serpent</th>
</tr>
</thead>
</table>

- Caveats:
  - The Twofish version optimised for bulk encryption is used throughout
  - A different version would perform much better at low block counts
  - Byte order conversion costs are omitted for CAST, Serpent & SAFER
AES Winners and Losers (IMHO)

• Should definitely go out on performance grounds:
  • DEAL, FROG, LOKI97, MAGENTA and SAFER+
• Should definitely stay in if secure:
  • MARS, RC6, Rijndael, Serpent and Twofish
• Should go out as a result of my personal bias:
  • HPC and DFC
• Undecided:
  • CAST, Crypton and E2