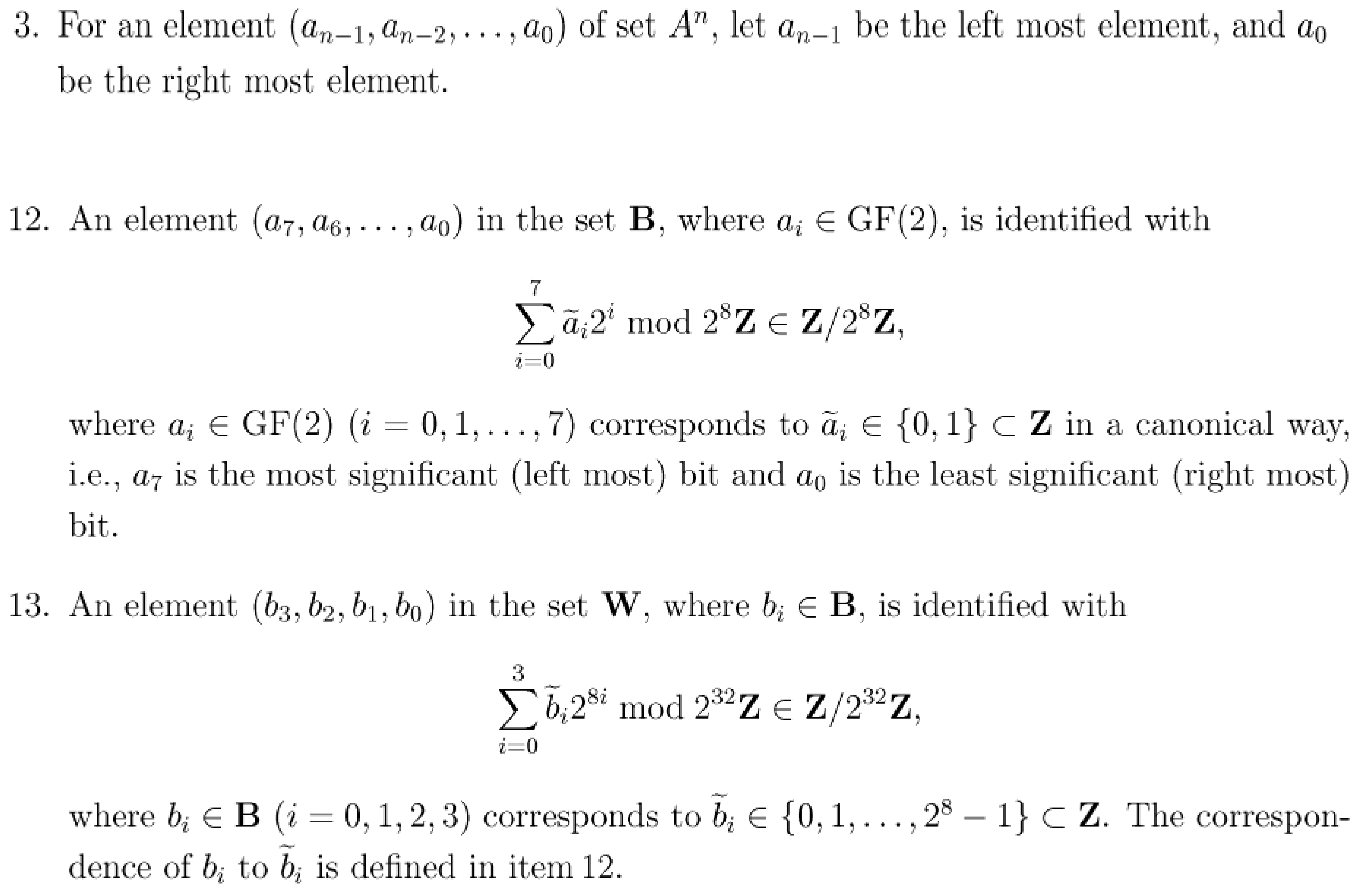
EXPERIENCE IN AES ALGORITHM IMPLEMENTATION

- 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 $\mathbf{B}$, where $a_i \in \mathrm{GF}(2)$, is identified with

$$\sum_{i=0}^{7} \tilde{a}_i 2^i \bmod 2^8\mathbf{Z} \in \mathbf{Z}/2^8\mathbf{Z},$$

where $a_i \in \mathrm{GF}(2)$ $(i = 0, 1, \ldots, 7)$ corresponds to $\tilde{a}_i \in \{0, 1\} \subset \mathbf{Z}$ 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.

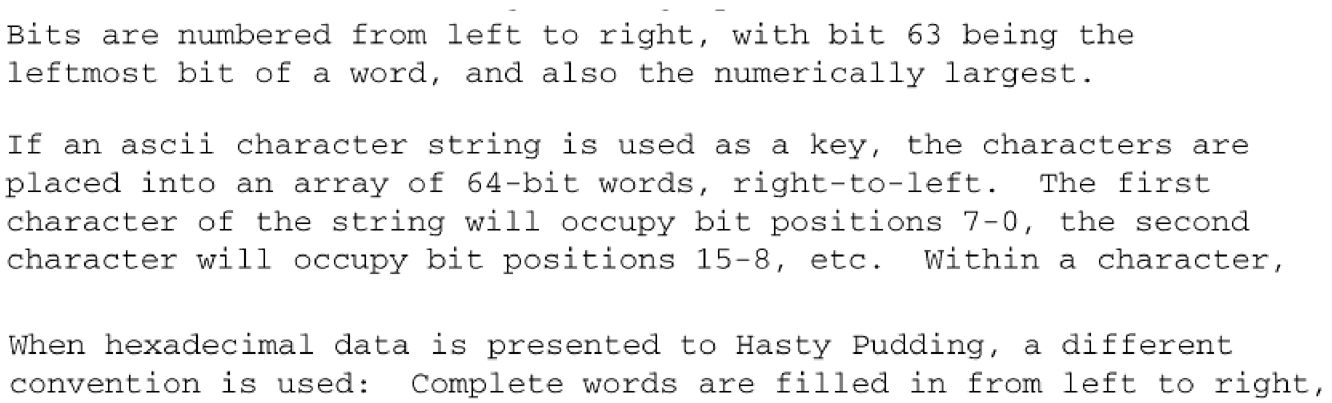
13. An element (b_3, b_2, b_1, b_0) in the set $\mathbf{W}$, where $b_i \in \mathbf{B}$, is identified with

$$\sum_{i=0}^{3} \widetilde{b}_i 2^{8i} \bmod 2^{32}\mathbf{Z} \in \mathbf{Z}/2^{32}\mathbf{Z},$$

where $b_i \in \mathbf{B}$ $(i = 0, 1, 2, 3)$ corresponds to $\widetilde{b}_i \in \{0, 1, \ldots, 2^8 - 1\} \subset \mathbf{Z}$. The correspondence of b_i to $\widetilde{b}_i$ is defined in item 12.

HPC and Serpent I/O Byte Order?

HPC:

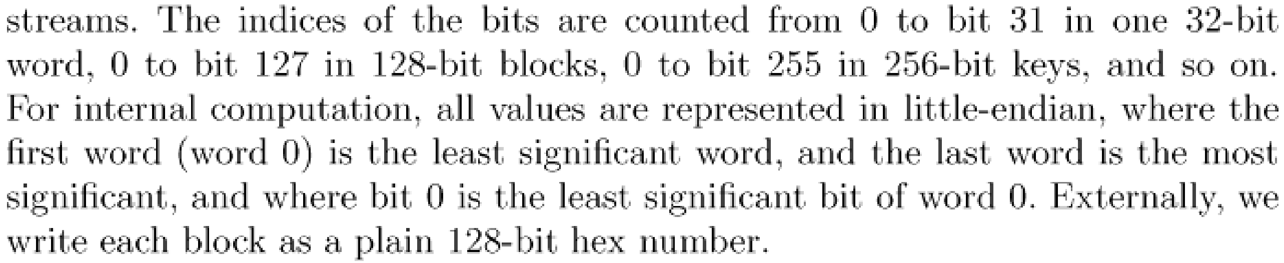


Bits are numbered from left to right, with bit 63 being the leftmost bit of a word, and also the numerically largest.

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

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

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;
}
```

10364 cycles

```
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);
}
```

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

```
#define byte(x,n)   *(((byte*)&x) + n)

...
store      register, long word [out]

load       register, byte [x + n]
….
```

```
#define byte(x,n)   ((byte)((x) >> (8 * n)))

...
store      register, long word [out]

load       register, long word [x]
r_shift    register, 8*n
….
```

- 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’

```
#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; t10 = t07 ^ t08;
t11 = t01 ^ t10; t12 = y   ^ t11;
t13 = b   & d  ; z   =     ~ t10;
x   = t13 ^ t12; t16 = t10 | x  ;
t17 = t05 & t16; w   = c   ^ t17;
}
```

```
#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

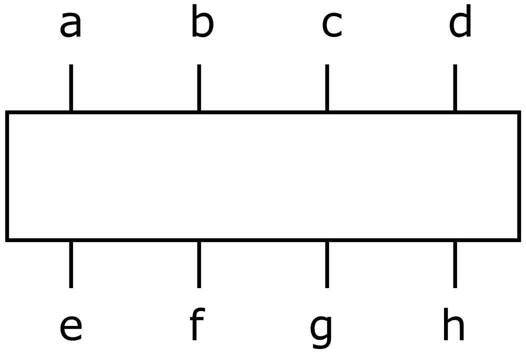
- With many C compilers the left hand code pulls in two cache blocks for EACH of the eight S boxes - 16 cache read/writes
- With the code on the right the whole encryption routine uses only two cache blocks
- This can improve Serpent speed by 10% or more

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:

Input	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Output	15	12	2	7	9	0	5	10	1	11	14	8	6	13	3	4

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



```
t0 = b xor (not a)
t1 = c xor (a or t0)
g = d xor t1
t2 = b xor (d or t0)
t3 = t0 xor g
h = t3 xor (t1 and t2)
t4 = t1 xor t2
f = h xor t4
e = t1 xor (t3 and t4)
```

- We want the ‘minimum cost’ circuit - the one with the fewest Boolean operations

The Serpent S Box Boolean Functions - II

```
t[-4] = 1
t[-3] = a
t[-2] = b    start list with 5
t[-1] = c    ‘primitive’ terms
t[0]  = d


t[1]  = t[-4] xor t[-3]
t[2]  = t[-2] ^ t[1]
t[3]  = t[-3] | t[2]
t[4]  = t[ 0] | t[2]
t[5]  = t[-1] ^ t[3]
   g  = t[ 0] ^ t[5]  - got one
t[7]  = t[-2] ^ t[4]
t[8]  = t[2] ^ g
t[9]  = t[5] & t[7]
   h  = t[8] ^ t[9]   - got two
t[11] = t[5] ^ t[7]
   f  = h ^ t[11]     - got three
t[13] = t[8] & t[11]
   e  = t[5] ^ t[13]  - got all four!
```

- 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

```
t[-4] = 1
t[-3] = a
t[-2] = b    start list with 5
t[-1] = c    ‘primitive’ terms
t[0]  = d


t[1]  = t[-4] xor t[-3]
t[2]  = t[-2] ^ t[1]
t[3]  = t[-3] | t[2]
t[4]  = t[ 0] | t[2]
t[5]  = t[-1] ^ t[3]
   g  = t[ 0] ^ t[5]  - got one
t[7]  = t[-2] ^ t[4]
t[8]  = t[2] ^ g
t[9]  = t[5] & t[7]
   h  = t[8] ^ t[9]   - got two
t[11] = t[5] ^ t[7]
   f  = h ^ t[11]     - got three
t[13] = t[8] & t[11]
   e  = t[5] ^ t[13]  - got four!
```

- 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

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

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

- 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)

Rijndael	CRYPTON	RC6	Serpent	MARS	CAST	SAFER	MAGENTA

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

RC6	Rijndael	MARS	Twofish	CRYPTON	CAST	E2	Serpent

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

RC6	MARS	Rijndael	Twofish	CRYPTON	CAST	E2	Serpent

- 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