A Software Implementation of 64-bit Bit slice Algorithm for AES

Madhu Venugopal
madhu.venugopal@cs.ucsb.edu

Skand S Gupta
skand@cs.ucsb.edu

University of California Santa Barbara

Introduction to AES

- Symmetric encryption and decryption.
- Block cipher, operates on block of 128 bits.
- Three different allowed key sizes: 128 bits, 192 bits or 256 bits.
- Number of rounds depends on key size.
  - 128 bit key: 10 rounds, 192 bit key: 12 rounds and 256 bit key: 14 rounds
- Assuming 128 bit keys for the following discussion and our implementation.
- The AES algorithm operates on a 4x4 matrix of bytes called state. The state undergoes a series of transformation.

AES Rounds

- Substitute Bytes. Substitutes each byte with a value from a look-up table called $Sbox$. The $Sbox$ entries are obtained by taking the inverse of each element in Galois Field, $GF(2^8)$.

- $ShiftRow$. Shifts each byte in the row by an offset.

- $MixColumn$. Multiplies each column by a constant matrix.

- $AddRoundKey$. Adds the round key which is derived from the initial key by using a key expansion algorithm.

- Round 1 only consists of $AddRoundKey$. Round 10 does not include $MixColumn$.

Case for Bit Slice Implementation of AES on Software

- Most efficient implementations are done on dedicated hardware engines such as in FPGAs and ASICs.
- Several applications such as networking software, OS modules need fast encryption but do not have hardware support.
- A naive software implementation is very slow.
- Bit slice algorithm performs N encryptions in parallel on a microprocessor with N-bit register width, resulting in significant performance boost.
- Bit slice implementation are immune to cache-timing attacks.

Bit Slice Implementation of AES

- Bit slice implementations convert the encryption algorithm into a series of logical bit operations using XOR, AND, OR and NOT logical gates.

- On a N-bit microprocessor, *bit slice* works on N inputs at a time called, *bundle* .

- On a 64-bite machine the *bundle* would contain 64 consecutive AES input blocks with each block occupying 2 words.

- The *bundle* is arranged so that the first bit of each input is present in the first word, the second bit on each input is present in the second word and so on.

- The re-arranged *bundle* is encrypted.

Bit Slice Implementation of AES

- All the encryption rounds are performed on the re-arranged bundle.
- The SBox table look-up used in $SubstituteByte$ is replaced with logical equations derived using composite field arithmetic.
- The final encrypted bundle is re-arranged at the end of encryption.

Bit Slice Implementation of AES - Input Bundle

- Bundle stored in memory for 64-bit processor

$b0_{63}$	..	$b0_4$	$b0_3$	$b0_2$	$b0_1$	$b0_0$
$b0_{127}$	..	$b0_{68}$	$b0_{67}$	$b0_{66}$	$b0_{65}$	$b0_{64}$
$b1_{63}$	..	$b1_4$	$b1_3$	$b1_2$	$b1_1$	$b1_0$
$b1_{127}$	..	$b1_{68}$	$b1_{67}$	$b1_{66}$	$b1_{65}$	$b1_{64}$
..	..	..	..	..	..	..
..	..	..	..	..	..	..
..	..	..	..	..	..	..
..	..	..	..	..	..	..
..	..	..	..	..	..	..
..	..	..	..	..	..	..
$b63_{63}$	..	$b63_4$	$b63_3$	$b63_2$	$b63_1$	$b63_0$
$b63_{127}$	..	$b63_{68}$	$b63_{67}$	$b63_{66}$	$b63_{65}$	$b63_{64}$

Bit Slice Implementation of AES - Input Bundle

- Rearranged bundle for 64-bit processor

$b63_0$	..	$b4_0$	$b3_0$	$b2_0$	$b1_0$	$b0_0$
$b63_1$	..	$b4_1$	$b3_1$	$b2_1$	$b1_1$	$b0_1$
$b63_2$	..	$b4_2$	$b3_2$	$b2_2$	$b1_2$	$b0_2$
$b63_3$	..	$b4_3$	$b3_3$	$b2_3$	$b1_3$	$b0_3$
..	..	..	..	..	..	..
..	..	..	..	..	..	..
..	..	..	..	..	..	..
..	..	..	..	..	..	..
..	..	..	..	..	..	..
..	..	..	..	..	..	..
$b63_{126}$	..	$b4_{126}$	$b3_{126}$	$b2_{126}$	$b1_{126}$	$b0_{126}$
$b63_{127}$	..	$b4_{127}$	$b3_{127}$	$b2_{127}$	$b1_{127}$	$b0_{127}$

Bit Slice - Our Implementation

- Based on the algorithms described in [2] and [1]
- Operating Environment: Open Suse 64-bit.
- Hardware: Intel x86 - 32 bit.
- 64-bit emulation via Vmware.

Bit Slice Implementation - Arranging the bundle

- The m^{th} bit from the word n is placed in the word n in the n^{th} bit of word m.
- The re-arrangement needs to be efficient.
- We use the Transpose algorithm described in [2]
- Complexity of the algorithm is $\Theta((n/2)log_2 n)$.
- The bundle is stored in a 128x64 bit matrix.

Bit Slice Implementation - Arranging the bundle

- The rearrangement requires that the bit m of an even row n be placed at the position $n/2$ of row m.

- If n is odd, the m^{th} bit of row n should be placed at the position $(n-1)/2$ of row $(63+m)$.

- The transpose of of all odd rows and even rows are calculated and the rows are re-arranged to put the $bundle$ in the required form.

Bit Slice Implementation of AES - Substitute Bytes

- The SBox table look-up is replaced with direct calculation of SBox using the sub-field arithmetic as described in [1]

- 2 main sub-steps in $SubstituteByte$ function:
  - Inverse. Let $c = a^{-1}$, the multiplicative inverse in $GF(2^8)$.
  - Affine transformation. Then the output is $s = Mc \oplus b$, where M is a specified 8x8 matrix of bits, b is a specified byte and the bytes c, b, s, are treated as vector of bits.

- Direct calculation of inverse (modulo an eighth-degree polynomial) of a seventh-degree polynomial is not easy. But calculation of the inverse (modulo a second-degree polynomial) of a first-degree polynomial is relatively easy.

Bit Slice Implementation of AES - Substitute Bytes

- Isomorphism between $GF(2^8)$and $GF(2^8)/GF(2^4)$to represent a general element g of $GF(2^8)$as a polynomial over $GF(2^4)$can be used.
- $GF(2^4)/GF(2^2)$can similarly be use to represent $GF(2^4)$.
- $GF(2^2)/GF(2)$is then used to represent $GF(2^2)$as linear polynomials over $GF(2)$.
- So finding an inverse in $GF(2^8)$can be broken down to inverse in $GF(2^4)$, which in turn can be broken down into $GF(2^2)$and finally $GF(2)$.

Bit Slice Implementation of AES - Substitute Bytes

- The state of AES engine after $SubstituteByte$ operation can be represented as:

$\mathsf{S}_{00}(B_{00} - B_{07})$	$\mathsf{S}_{04}(B_{32} - B_{39})$	$\mathsf{S}_{08}(B_{64} - B_{71})$	$\mathsf{S}_{12}(B_{96} - B_{103})$
$\mathsf{S}_{01}(B_{08} - B_{15})$	$\mathsf{S}_{05}(B_{40} - B_{47})$	$\mathsf{S}_{09}(B_{72} - B_{79})$	$\mathsf{S}_{13}(B_{104} - B_{111})$
$\mathsf{S}_{02}\ (B_{16} - B_{23})$	$\mathsf{S}_{06}(B_{48} - B_{55})$	$\mathsf{S}_{10}(B_{80} - B_{87})$	$\mathsf{S}_{14}(B_{112} - B_{119})$
$\mathsf{S}_{03}\ (B_{24} - B_{31})$	$\mathsf{S}_{07}(B_{56} - B_{63})$	$\mathsf{S}_{11}(B_{88} - B_{95})$	$\mathsf{S}_{15}(B_{120} - B_{127})$

- Each element S_n consists of 8 words, B_{8n} to B_{8n+7}.

- Each word is of N bits representing the N encryptions taking place in parallel.

Shift Row and Mix Column

- The $ShiftRow$ operation shifts the second row left by eight bits, the third row by sixteen bits and fourth row by twenty four bits as shown below:

$S_{00}(B_{00} - B_{07})$	$S_{04}(B_{32} - B_{39})$	$S_{08}(B_{64} - B_{71})$	$S_{12}(B_{96} - B_{103})$
$S_{05}(B_{40} - B_{47})$	$S_{09}(B_{72} - B_{79})$	$S_{13}(B_{104} - B_{111})$	$S_{01}(B_{08} - B_{15})$
$S_{02}(B_{80} - B_{87})$	$S_{14}(B_{112} - B_{119})$	$S_{02}(B_{16} - B_{23})$	$S_{06}(B_{48} - B_{55})$
$S_{03}(B_{120} - B_{127})$	$S_{03}(B_{24} - B_{31})$	$S_{07}(B_{56} - B_{63})$	$S_{11}(B_{88} - B_{95})$

- $MixColumn$ is essentially multiplication of each column of the matrix with a permutation of [2 3 1 1]

- For example $MixColumn$ output for the first byte is given by:
  $S^{'}_{00} = 2S_{00} + 3S_{05} + S_{10} + S_{15}$

Bit Slice Implementation of AES - Key Schedule

- The round keys also need to go through the same transformation as the input $bundle$.
- Each round key is repeated 64 times.
- Same transpose is applied to round keys as the input $bundle$.
- The $AddRoundKey$ adds each word in the bundle with the corresponding word in the key $bundle$.

Evaluation

- We compare the number of clock ticks required to encrypt 128 bundles in our implementation vs OpenSSL AES implementation. As baseline we compare against the encryption times in [2] for 64-bit size $bundle$ on a Core 2. We refer to this implementation as RSD (for author's initials).

- The OpenSSL implementation is a 32 bit software implementation of AES. Note that this implementation is optimized for 32 bit environment and should perform well on our test environment. Hence it provides a good reference implementation to compare with.

Evaluation

Table 1: The Test Environment

Operating System	Open Suse 64-bit
Microprocessor	Intel Core 2 duo (64 bit emulation using Vmware)
Core Speed	2.0 GHz
Memory	512 Mb
Compiler	gcc-4.3.1
Input $bundle$	64-bits
Key Size	128-bits

Table 2: Encryption Times

	Our Implementation	OpenSSL AES	RSD Bit Slice
Clock ticks for 128 bundles	47400	39000	-
Clock ticks per bundle	370	304	302

Conclusion

- Our implementation does almost as good as RSD implementation even though we run tests over Vmware and with only 512 Kb of memory (RSD uses 4Gb of memory).

- The OpenSSL AES does better, presumably because it is a 32 bit implementation and the underlying hardware in our test environment is 32 bit as well.

References

References

[1] Canright D. *A Very Compact Rijndael S-box*. Montrery, CA, 2004.

[2] Devi A.S.L Rebeiro Chester, Selvakumar David. *Bitslice Implementation of AES*. Springer Berlin, 2006.