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 \textit{Sbox}. The \textit{Sbox} entries are obtained by taking the inverse of each element in Galois Field, \( GF(2^8) \).

- \textit{ShiftRow} . Shifts each byte in the row by an offset.

- \textit{MixColumn} . Multiplies each column by a constant matrix.

- \textit{AddRoundKey} . Adds the round key which is derived from the initial key by using a key expansion algorithm.

- Round 1 only consists of \textit{AddRoundKey} . Round 10 does not include \textit{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-bit 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

<table>
<thead>
<tr>
<th>b0_{63}</th>
<th>..</th>
<th>b0_{4}</th>
<th>b0_{3}</th>
<th>b0_{2}</th>
<th>b0_{1}</th>
<th>b0_{0}</th>
</tr>
</thead>
<tbody>
<tr>
<td>b0_{127}</td>
<td>..</td>
<td>b0_{68}</td>
<td>b0_{67}</td>
<td>b0_{66}</td>
<td>b0_{65}</td>
<td>b0_{64}</td>
</tr>
<tr>
<td>b1_{63}</td>
<td>..</td>
<td>b1_{4}</td>
<td>b1_{3}</td>
<td>b1_{2}</td>
<td>b1_{1}</td>
<td>b1_{0}</td>
</tr>
<tr>
<td>b1_{127}</td>
<td>..</td>
<td>b1_{68}</td>
<td>b1_{67}</td>
<td>b1_{66}</td>
<td>b1_{65}</td>
<td>b1_{64}</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>b63_{63}</td>
<td>..</td>
<td>b63_{4}</td>
<td>b63_{3}</td>
<td>b63_{2}</td>
<td>b63_{1}</td>
<td>b63_{0}</td>
</tr>
<tr>
<td>b63_{127}</td>
<td>..</td>
<td>b63_{68}</td>
<td>b63_{67}</td>
<td>b63_{66}</td>
<td>b63_{65}</td>
<td>b63_{64}</td>
</tr>
</tbody>
</table>
- Rearranged bundle for 64-bit processor

<table>
<thead>
<tr>
<th>b63_0</th>
<th>..</th>
<th>b4_0</th>
<th>b3_0</th>
<th>b2_0</th>
<th>b1_0</th>
<th>b0_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>b63_1</td>
<td>..</td>
<td>b4_1</td>
<td>b3_1</td>
<td>b2_1</td>
<td>b1_1</td>
<td>b0_1</td>
</tr>
<tr>
<td>b63_2</td>
<td>..</td>
<td>b4_2</td>
<td>b3_2</td>
<td>b2_2</td>
<td>b1_2</td>
<td>b0_2</td>
</tr>
<tr>
<td>b63_3</td>
<td>..</td>
<td>b4_3</td>
<td>b3_3</td>
<td>b2_3</td>
<td>b1_3</td>
<td>b0_3</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>b63_126</td>
<td>..</td>
<td>b4_126</td>
<td>b3_126</td>
<td>b2_126</td>
<td>b1_126</td>
<td>b0_126</td>
</tr>
<tr>
<td>b63_127</td>
<td>..</td>
<td>b4_127</td>
<td>b3_127</td>
<td>b2_127</td>
<td>b1_127</td>
<td>b0_127</td>
</tr>
</tbody>
</table>
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.
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 all odd rows and even rows are calculated and the rows are re-arranged to put the bundle in the required form.
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 used 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 $GF(2)$. 
The state of AES engine after \textit{SubstituteByte} operation can be represented as:

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

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.
• The \textit{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:

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

• \textit{MixColumn} is essentially multiplication of each column of the matrix with a permutation of \([2 \ 3 \ 1 \ 1]\)

• For example \textit{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.
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

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Open Suse 64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microprocessor</td>
<td>Intel Core 2 duo (64 bit emulation using Vmware)</td>
</tr>
<tr>
<td>Core Speed</td>
<td>2.0 GHz</td>
</tr>
<tr>
<td>Memory</td>
<td>512 Mb</td>
</tr>
<tr>
<td>Compiler</td>
<td>gcc-4.3.1</td>
</tr>
<tr>
<td>Input bundle</td>
<td>64-bits</td>
</tr>
<tr>
<td>Key Size</td>
<td>128-bits</td>
</tr>
</tbody>
</table>

Table 2: Encryption Times

<table>
<thead>
<tr>
<th></th>
<th>Our Implementation</th>
<th>OpenSSL AES</th>
<th>RSD Bit Slice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock ticks for 128 bundles</td>
<td>47400</td>
<td>39000</td>
<td>-</td>
</tr>
<tr>
<td>Clock ticks per bundle</td>
<td>370</td>
<td>304</td>
<td>302</td>
</tr>
</tbody>
</table>
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
