Software Implementation of OCB Mode

Baba Prashanth Namatheertham, Ritesh Suppala, Bharath Govindool, Paradesi Ravi Kiran

Abstract—The OCB (Offset Code Book) is a block cipher mode of operation used for encryption which provides both confidentiality and authenticity at the same time. This implementation is cost efficient as we do not have to design a separate hardware for authentication process. The OCB can be executed on any platform like C, JAVA and Assembly languages. This implies that it can be implemented by hardware or software.

Index Terms—OCB, Rijndael AES

I. INTRODUCTION

The new modes of operation have been introduced in order to enhance the functionality of the block cipher encryptions. The OCB mode is one of them. The OCB mode is an authenticated encryption scheme. This OCB scheme has been invented by Phil Rogaway. The OCB mode is implemented in Hybrid mode. The main code is written in C and then it is interfaced to the hardware [8].

In our present work the OCB has been implemented in C (Software). In general the authenticity is provided by a shared secret key encryption on a hash function (Message Authentication Code). The implementation of this code requires another algorithm and sufficient hardware. This increases the cost of the overall encryption system. In the OCB mode, the algorithm is designed in such a way that it provides both authenticity and confidentiality.

In the OCB mode any block cipher such as DES or AES can be used. In our project we have used the Advanced Encryption Standard. To be more precise it is Rijndael AES. In this encryption standard each time 128bits of message are considered and pre computations are done on it and are encrypted.

In our present work OCB mode is implemented in the same way as it was created [5] except for that the mode is implemented in software. By doing so we can create a code which will allow us to use the encryption method in our personal PCs where data privacy is necessary in communication. The OCB mode can be shown as an efficient method compared to other modes which is done in this project. The other mode used for the comparison is the CBC mode which is also implemented in C. The clock cycles required for the execution of the code are calculated in both modes and are compared. From these values efficiency of the code can be confirmed.

II. THE OCB MODE

Before running the OCB mode certain parameters need to be initialized. The parameters include the offset generation which requires L value generation, Random number generation and also padding. Thus the overall implementation of the OCB mode involves the following steps.

1. Partitioning the entire text message into blocks of 128 bits.
2. The last message may not contain exactly 128 bits so it is padded with zeroes.
3. A random number generation and encrypting it with AES
4. L value generation, the number of L values is equal to the number of blocks of message.
5. The generation of the hash function, this is used for message authentication.
6. Now by using all the above values the message blocks are encrypted using AES.

The figure 1 shows the outline model of OCB.
In order to implement the OCB mode the first major task is to understand the encryption method used which is the Advanced Encryption Standard (AES). Hence first and foremost the AES encryption scheme is studied clearly. The Rijndael AES encryption method [1] [3] operates on 128 bits at a time and the size of the key can be in 128, 192 or 256 bits. The key used for the AES scheme is the key for the entire OCB mode. The AES cipher exhibits greater complexity and more secure compared to DES block cipher. Hence this cipher is preferred to DES.

The AES scheme involves certain number of rounds which can be 10, 12 or 14 basing on the key size. In our implementation we consider a key size of 128 bits so the number of rounds are 10 [1]. Each round performs the following operations.

1. Substitute Bytes
2. Shift Rows
3. Mix Columns
4. Add Round Key

This procedure is done for 10 rounds and for each round a new round key is used. If we arrange the 128 bits in terms of bytes and put them in a matrix then we get a 4x4 matrix of four columns of words. Thus each 128 bit data represents four words arranged in columns. Before the round operations are executed there are few initial operations to be carried out. The key which is a 4x4 matrix consisting of 4 words (128 bits) key is expanded into a 44 word key. For every round 4 words are used as a round key. The Fig 2 explains the AES encryption. The given plain text is first arranged in the form of bytes in a 4x4 matrix (4 words) and the given key is expanded into a 44 word key.

**Substitute Bytes:** In order to perform this substitution first we need to initialize the substitution box. This substitution box is a 16x16 matrix. Each byte of the message is substituted with another byte from the S-Box. The byte is selected from the S-Box using the row and column numbers. The row is given by the first four bits of the message byte and
the next four bits represent the column. Basing on this logic the bytes in the message are substituted. The S-Box values are generated basing on certain rules [3].

**Shift Rows:** The Shift rows block of the round will shift the bytes in the rows. The row byte elements are not shifted. The second row byte elements are shifted by one byte towards the left, the third row byte elements are shifted by two bytes towards left and the final fourth row byte elements are shifted by three bytes towards left. This operation will shift all the rows.

**Mixing Columns:** This transformation operates on each column individually. Each byte of a column is mapped into a new value that is a function of all the four bytes in that column. Each element in the product matrix is the sum of products of elements of one row and one column. In order to do this we first consider each word in the 4 word message and perform the transformation. The matrix multiplication performed as mention in [2].

**Add Round Key:** The Add Round Key is a simple xor operation where the message block after the transformation of mixing columns is xored with the round key which is available for each round. The round key is different for different rounds.

**Key Expansion:** The first 4 words of the expanded key are just the 4 words of the input key. For the remaining 40 columns the word is a function of preceding column and four preceding column. If the number of the column is a multiple of 4 then the column is a function of four preceding column and a function of preceding column (g). This function g first shifts the word towards left, performs byte substitution using the S-box and finally xor’s the byte value with Rcon[j] array which are predefined for a particular round. By using this g function the columns of multiple four are expanded. Thus the whole 4 word key is expanded to a 44 word key.

As shown in Fig 2 the AES encryption follows the mentioned steps. The plain text message should be of 4 word or 128 bits only. Hence the given plain text is divided into the blocks of 128 bits or 4 words.

**III. THE OCB IMPLEMENTATION**

**First Step:** The five steps mentioned earlier are to be first implemented using software. The first step is partitioning the entire message in the input file into blocks of 128 bits.

**Second Step:** The last message block may not contain 128 bits so it is padded with zeroes. The number of zeroes that can be padded are from 1 to 127.

**Third Step:** A random number is generated using the C compiler. This can be done by using special functions included in the header `<math.h>`. The random number is encrypted using the AES to generate the NONCE. This NONCE is used to generate the offset value. This value along with the L value will give the offset.

**Fourth Step:** One of the important features of the OCB mode is the L value. This L value is different for different message blocks. This will increase the complexity of the encryption method.

1. In order to generate the L values initially 128 bits of 0 are taken and encrypted with the AES cipher
2. The total number of L values generated is equal to the number of message blocks. Let the encrypted value be equal to L[0].
3. The first L value L[-1] is generated as follows

   - If the MSB of the encrypted value L0 is 0 then L[-1] = L[0]>>1.
   - If the MSB of L0 is 1 then L1=L0>>1 and the last byte of the L0 value is xored with 87H
4. The next L values L2, L3, etc are generated using the following logic.
- If MSB of L[i]=0 then L[i+1]=L[i]<<1.
- If MSB of L[i]=1 then L[i+1]<<1 and the last byte of L[i] is xored with 83H and the first byte with 80H

Thus using these L values the offset is generated. For each offset the L value is xored with the NONCE. Hence for the message block i the offset is going to be L[i] NONCE. This offset value is xored with the message before and after the encryption. The offset is referred as in Phil Rogway’s paper [9].

In generation of the L values, multiplication is involved which is performed basing on the polynomial multiplication of binary numbers. The OCB uses the gray code which makes computations of the successive points easy. The Gray code can be generated by using the following algorithm.

\[
\text{for } i=1 \text{ to } m \\
L[i] = L[i-1] \oplus \text{ntz}(i)
\]

where \(\text{ntz}(i)\) is the number of zeroes trailing when the number i is represented in binary.

---

**Figure 3: OCB Implementation**

\[\text{Cipher} = C[1] \ C[2] \ldots \ldots C[m] \ \text{Tag}\]
Fifth Step: The tag value is generated by first generating a hash function. A hash function is a digest of the original message. The hash function in our case is a check sum formed by xoring all the message values to each other. Except for the final block of message all message blocks are xored. Instead of xoring the final message block directly we xor the Cipher text of the message block. This check sum is xored with the final offset value $L[m]$ and when encrypted with AES results in the Tag value which is used for authentication.

Sixth Step: The final step of the entire process is to encrypt all the value and generate the cipher text with the tag value.

The Fig.3 shows the entire OCB mode in detail.

IV. ALGORITHM FOR OCB

AES (K) (AES encryption with key K 128 bits)
NONCE (R) = AES (Random number, K).
Offset = AES (R xor L, K).

for $i = 0$ to $m-1$.

M[i] = M[i] xor offset
C[i] = AES (M[i], K)
C[i] = C[i] xor offset

X[m] = Len [M (m)] xor L (-1)
X[m] = M[m] xor offset
Y[m] = AES (M[m], K)
C[m] = Y[m] xor M[m]

for $i = 0$ to $m-1$

Hash = Hash xor M [m-1] xor C[m] xor Y[m]

Tag = AES (Hash, K)
Cipher = C [1] C [2]……C[m] Tag

V. DISCUSSION

OCB has been designed to have a variety of desirable properties [5]. The following discussions explain few of the properties.

Arbitrary-length messages and minimal cipher text expansion: One of the key characteristics of OCB is that any string $M$ can be encrypted, and doing this yields a cipher text $C$ having $|M|+t$ bits. That is, the length of the “cipher text core” the portion $C = C[1] · · · C[m]$ of the cipher text that excludes the tag is the same as the length of the message $M$. This is better, by up to $m$ bits, than what one gets with conventional padding. But we do not regard the nonce as part of the cipher text. Including it, the amount of information that needs to be sent to the receiver is $|M| +t+n$ bits, where bits are used to communicate the nonce $R$. The value of $n$ could be anything in $[0 . m]$, depending on the application.

Single block-cipher key: OCB makes use of just one block-cipher key, K. While $L = EK(0m)$ functions rather like a key and would normally be computed at key-setup time, and while standard key-separation techniques can always be used to obtain many keys from one, the point is that, in OCB, all block-cipher invocations use the one key K. Thus only one block-cipher key needs to be setup, saving on storage space and key-setup time.

Weak nonce requirements: We believe that modes of operation that require a random IV are often misused. As an example, consider CBC mode [1], where $C[i] = EK(M[i] xor C[i-1])$ and $C[0] = IV$. Many standards and many books (e.g., Schneier, Applied Cryptography, 2nd edition, p. 194) suggest that the IV may be a fixed value, a counter, a timestamp, or the last block of cipher text from the previous message. But if it is any of these things one certainly will not achieve any of the standard definitions of privacy [Bellare et al. 1997; Goldwasser and Micali 1984]. It is sometimes suggested that a mode which needs a random IV is preferable to one that needs a nonce: it is said that state is needed for a nonce, but not for making random bits. We find this argument wrong. First, a random value of sufficient length can always be used as a nonce, but a nonce cannot be used as a random value. Second, the manner in which systems provide “random” IVs is invariably stateful anyway: unpredictable bits are too expensive to harvest for each IV, so one does this rarely, using
state to generate pseudorandom bits from unpredictable bits harvested before. Third, the way to generate pseudorandom bits needs to use cryptography, so the prevalence of non-cryptographic pseudorandom number generators routinely results in implementation errors. Fourth, nonce-based schemes facilitate simple replay-detection. Finally, nonces can be communicated using fewer bits than random values.

On-line: OCB encryption is on-line [8]: one can output a stream of cipher text bits as a stream of plaintext bits arrive, the output stream having constant latency and the transformation using constant memory. When one receives an indication that the plaintext is over, the final chunk of cipher text is output. One need not know the length of the plaintext in advance of processing it. This allows the efficient encryption of strings whose representation uses a special character (e.g., a zero byte) to indicate the string’s end. An incremental interface (in the style popular for cryptographic hash functions) could be used to support this functionality. OCB decryption is likewise on-line, but with an important difference: one can produce a stream of plaintext bits as the stream of cipher text bits comes in, but when the cipher text stream is finished one may need to “cancel” the plaintext stream that has issued (having found the cipher text to be invalid). In such a case, nothing about the cipher text (like what was the canceled plaintext) should be adversarially available beyond an indication of its invalidity. In any authenticated encryption scheme, decryption can be on-line only to this extent.

Significance of being efficient: Shaving off a few block-cipher calls or a few bytes of cipher text may not seem important. But often one is dealing with short messages; for example, roughly a third of the messages on the Internet backbone are 43 bytes. If one is encrypting messages of such short lengths, one should be careful about message expansion and extra computational work since, by percentage, the inefficiencies can be large. The argument has been made that making a major effort to save a factor of two in computational efficiency is marginal in the first place: “Moore’s law” will soon deliver such an improvement anyway, by way of faster hardware. Along with processors getting faster, security has become increasingly an issue, and low-power and embedded processors have become more prevalent. The result is a need to cryptographically process more and more data, and often by “dumb” execution vehicles. Hardware advances have changed our understanding of what efficiency entails but, to date, hardware advances have not made cryptographic efficiency less important.

Optional pre-processing: Implementations can choose how many L(i) values to precompute [5]. Since only one block-cipher call is needed to compute all of the L(i) values, plus a shift and a conditional xor for each value, it is feasible to do no preprocessing: OCB is appropriate even when each session is a single, short message.

Provable security: Provable security has become a popular goal for practical protocols. This is because it provides the best way to gain assurance that a cryptographic scheme does what it is should. For a scheme that enjoys provable security one does not need to consider attacks on the scheme, since successful ones imply successful attacks on some simpler object. When we say that “OCB is provably secure” we are asserting the existence of two theorems. One says that if an adversary A could do a good job at forging Cipher texts with OCB [E, t ] then there would be an adversary B that does a good job at distinguishing (E(K), E^-1(K)), for a random key K for a random permutation. The other theorem says that if an adversary A could do a good job at distinguishing OCB [E, t] encrypted messages from random strings, and then there would be an adversary B that does a good job at distinguishing E(K), for a random key K for a random permutation. Theorems of this sort are called reductions. In cryptography, provable security means giving reductions (along with the associated definitions). Provable security begins with Goldwasser and Micali [Goldwasser and Micali 1984]. The style of provable security that we use here—where the primitive is a block cipher, the scheme is a mode of operation, and the analysis is concrete (no asymptotics)—is the approach of Bellare and Rogaway [Bellare et al. 1997; Bellare
et al. 1995; Bellare et al. 2000]. It is not enough to know that there is a provable-security result; one should also understand the definitions and the bounds. We have already sketched the definitions. When we speak of the bounds we are addressing “how effective is the adversary B in terms of the efficacy of adversary A” (where A and B are as above). For OCB, the bounds can be roughly summarized as follows. An adversary can always forge with probability $1/(2^t)$ . Beyond this, the maximal added advantage is at most $(p^2)/2m$, where $p$ is the total number of blocks the adversary sees. The privacy bound likewise degrades as $(p^2)/2n$. The conclusion is that one is safe using OCB as long as the underlying block cipher is secure and _ is small compared to $2^t (m/2)$. This is the same security degradation one observes for CBC encryption and in the bound for the CBC MAC [Bellare et al. 1997; Bellare et al. 2000]. This kind of security loss was the main motivation for choosing a block length for AES of $n = 128$ bits.

**Simplicity:** Simplicity has been a central design goal. Some of OCB’s characteristics that contribute to simplicity are: (1) Short and full final-message-blocks are handled uniformly, not splitting into separate cases. (2) Only the simplest form of padding is used: append a minimal number of 0-bits to make a string whose length is a multiple of $n$. This method is computationally fastest and helps avoid a proliferation of cases in the analysis. (3) Only one algebraic structure is used throughout the algorithm: the finite field GF $(2^m)$. (4) In forming the sequence of offsets, Gray-code coefficients are taken monotonically, starting at 1 and stopping at $m$. One never goes back to an earlier offset or forms more offsets than there are blocks.[6]

**Not fixing how the nonce is communicated:** We do not specify how the nonce is chosen or communicated. Formally, it is not part of the cipher text (though the receiving party needs it to decrypt). In many contexts, there is already a natural value to use as a nonce (e.g., a sequence number already present in a protocol flow, or implicit because the parties are communicating over a reliable channel). Even when a protocol is designed from scratch, the number of bits needed to communicate the nonce will vary. In some applications, 32 or even 8 bits is enough. For example, one might have reason to believe that there are at most 232 messages that will flow during the connection, or one may communicate only the lowest 8 bits of a sequence number, counting on the receiver to anticipate the high-order bits.

**Not fixing the tag length:** The number of bits necessary for the tag varies according to the application [5]. In a context where the adversary obtains something quite valuable from a successful forgery, one may wish to choose a tag length of 80 bits or more. In contexts such as authenticating a video stream, where an adversary would have to forge many frames to have a major impact on the image, an 8-bit tag may be appropriate. With no universally correct value to choose, it is best to leave this parameter unspecified. Short tags seem to be more appropriate for OCB than for some other MACs, particularly Carter-Wegman MACs. Many Carter-Wegman MACs have the property that if you can forge one message with probability $e$ then you can forge an arbitrary set of (all correct) messages with probability $e$. This does not appear to be true for OCB, though we have not investigated formalizing or proving such properties.

**Forming R using a block-cipher call:** During our work we discovered that there are methods for authenticated-encryption that encrypt $M$ using $[|M|/m]+1$ block-cipher calls, as opposed to our $[|M|/m] + 2$ calls. Shai Halevi has also made this finding [Halevi 2001]. However, the methods we know to shave off a block-cipher call either require an unpredictable IV instead of a nonce, or they add conceptual and computational complexity to compute the initial offset $R$ by non-cryptographic means (e.g., using a finite-field multiplication of the nonce and a key variant).

**Definition of the checksum:** An initially odd-looking aspect of OCB’s definition is the definition of Checksum = $M[1] \oplus \cdots \oplus M[m - 1] \oplus C[m] \oplus 0 \oplus Y[m]$. In Jutla’s scheme, where one assumes that all messages are a positive multiple of the block length, the checksum is the simpler-looking $M[1] \oplus \cdots \oplus M[m - 1] \oplus M[m]$. We
comment that these two definitions are identical in the case that $|M[m]| = m$. What is more, the definition $\text{Checksum} = M[1] \oplus \ldots \oplus M[m-1] \oplus M[m] \oplus 0$ turns out to be the wrong way to generalize the Checksum to allow for short-final-block messages; in particular, the scheme using that checksum is easily attacked.

VI. PROPERTIES OF OCB

Basing on the above discussion we can write the following properties of the OCB mode.

1. OCB is an authenticated-encryption scheme: encrypted messages are both private and authenticated provided by the AES encryption and Tag value.
2. Very strong forms of privacy are achieved. These strong properties make OCB easier to correctly use in protocols than standard privacy modes.
3. It has ever faster implementations as machines offer up more and more parallelism, and it is good for encrypting messages in hardware at the highest network speeds.
4. OCB works with any block cipher.
5. OCB avoids the need for a random IV (a nonce is enough).
6. OCB uses only a single block-cipher key.
7. Key setup in OCB is very easy. (Typically one block-cipher calls, plus a few shifts and conditional xors).
8. OCB doesn't need much memory to run (and a memory-stingy implementation doesn't give up much speed).
9. OCB doesn't need much memory to run (and a memory-stingy implementation doesn't give up much speed).
10. OCB can encrypt messages of any bit length. Messages don't have to be a multiple of the block length, and no separate padding regime is needed.
11. Messages of all lengths are treated in a single, uniform manner.
12. The length of an OCB ciphertext is the same as the length of the plaintext (discounting the nonce and the "tag"). In particular, no bits are wasted due to padding.
13. OCB is simple to understand and implement. It uses GF $(2^{128})$ arithmetic and a Gray code [5], but it all comes down to some xors and shifts. One doesn't have to understand the mathematics to implement the scheme.
14. OCB is provably secure. It provably meets its goals; as long as the underlying block cipher AES meets standard cryptographic assumptions.

VII. COMPARISION OF OCB WITH OTHER MODE

From the above discussion we can say that OCB is a very efficient way of encryption. The very important feature is that it provides parallelism. The other mode implemented for comparison is a CBC mode. In this mode initially we assume a value IV and a message which is a multiple of block length uses $M/m$ block cipher calls. If this basic CBC mode is included with the MAC then the number of calls increases to $2(M/m) + 1$ calls. As compared with any mode OCB has an overhead beyond the block cipher calls. Per block this overhead is about $m$ xor operations plus the associated logic. The associated logic depends on the generation of the L values etc.

All of the L values when computed on the fly is inefficient so they are precomputed. Starting with 128 bits of 0s and shifting it and xorring it and so forth.

In our present work we have introduced the function in order calculate the Clock time. This will help us in comparing the time taken for encryption for the two modes. From the observations we can conclude that OCB is only a slight overhead over a normal AES encryption. For the same key a 1kbyte
file is encrypted with OCB and with CBC. The clock cycles are noted. The following table gives the values.

**Table 1**: Comparison of OCB and CBC modes

<table>
<thead>
<tr>
<th>INDEX</th>
<th>OCB Mode</th>
<th>CBC Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6900 clock tics</td>
<td>1400 clock tics</td>
</tr>
<tr>
<td>2</td>
<td>13000 clock tics</td>
<td>1800 clock tics</td>
</tr>
<tr>
<td>3</td>
<td>16000 clock tics</td>
<td>2000 clock tics</td>
</tr>
<tr>
<td>4</td>
<td>22000 clock tics</td>
<td>2900 clock tics</td>
</tr>
<tr>
<td>Mean</td>
<td><strong>14475 clock tics</strong></td>
<td><strong>2025 clock tics</strong></td>
</tr>
</tbody>
</table>

From the above table it is clear that OCB mode takes more clock cycles than the CBC mode.

**VIII. THE C COMPILER**

In the implementation process we have used C as our platform as C provides many functions which will help us in executing the mode easily. In C the operations on hexadecimal values is very convenient. All the computations need to be done in the hexadecimal values. Though C offers many features the code is very lengthy. Few operations like shifting of 128 bits need some extra logic. The main difficulty is performing the mixing of columns in the AES encryption. C also provides functions to calculate the system clock cycles by using the header `<time.h>` . Another important feature in C is debugging where we can stop the execution at a particular point by using a watch signal and analyze the values of the variables. By considering all the above facts we used C as our platform for implementing the mode.

**IX. CONCLUSION**

Thus we can conclude that OCB is a very efficient implementation. Though the clock cycles show a significant difference the execution time is almost same and varies in the power of milli seconds. There is no much difference in the code execution between the two modes. The properties of OCB have been mentioned. It is clear that the OCB mode will provide both authentication and privacy at the same time efficiently.

**X. REFERENCES**


**Baba Prashanth Namatheertham**, Paradesi Ravi Kiran, Bharath Govindool, Ritesh Suppala are all graduate students in the department of Electrical Engineering, George Mason University.