Efficient Error-Propagating Block Chaining.

Conference Paper · December 1997

DOI: 10.1007/BFb0024478 · Source: DBLP

2 authors, including:

André Zúquete
University of Aveiro
97 PUBLICATIONS **632** CITATIONS

SEE PROFILE

Efficient Error-Propagating Block Chaining

André Zúquete and Paulo Guedes

IST / INESC
R. Alves Redol 9, 1000 Lisboa, PORTUGAL
(Andre.Zuquete, Paulo.Guedes)@inesc.pt

Abstract. This document presents EPBC, Efficient Error-Propagating Block Chaining, a new and efficient block encryption mode using both plaintext and ciphertext feedback. This encryption mode is similar to another one, IOBC, and was likewise designed to propagate erroneous decryptions of tampered blocks of ciphered data to all following blocks, hence allowing to validate the integrity of that data using a predefined trailing value. However, EPBC is more secure than IOBC, as it is not vulnerable to any known-plaintext attacks, and is more efficient than IOBC. Performance tests ran on a SPARCstation 10/40 show that EPBC is in average 1.2 times faster than IOBC, and 6.3 to 10.9 times faster than a common combination of an encryption mode and a one-way hash function (CBC and MD5).

1 Introduction

The integrity control of encrypted data requires data to carry an extra integrity control value. This integrity control value allows legitimate principals to detect, after decryption, modifications on the original data contents. There are two basic ways to handle integrity control values:

1. They are generated using a one-way hash function (e.g. MD5 [9] or SHA [11]) from all the bits of the original or recovered plaintext data independently of the encryption/decryption algorithm [13, 3, 2, 1].
2. They are predefined values, which may be set up in many different ways: agreed between interacting peers, derived from a secret value, like the encryption key, or efficiently computed from some bits, but not necessarily all, of the plaintext data. In this case the encryption/decryption algorithm must guarantee that any modifications of the ciphertext will propagate erroneous decryptions until the end of the ciphertext, thus affecting the resulting integrity control value [12, 14].

The second way to handle the integrity control of encrypted data is attractive because one may save the time expended in the generation of data's hash values by slightly increasing the complexity of the encryption mode. However, most commonly used block encryption modes, like Electronic Code Book (ECB) or Cipher Block Chaining (CBC) [4], do not propagate erroneous decryptions of a modified ciphertext block to all following blocks. There are several examples of

encryption modes providing error propagation, like the Kerberos' Propagating CBC [12]. Unfortunately, they have weaknesses, such as allowing the addition of arbitrary values to ciphertext blocks, swapping of ciphertext blocks, or the replacement of ciphertext blocks by new ones using known-plaintext attacks.

This document presents the Efficient Error-Propagating Block Chaining (EPBC), a new encryption mode providing error propagation without suffering from the weaknesses of other encryption modes with a similar functionality. In particular, it resists to all the attacks previously referred. Note that EPBC is not intended to be used as a keyed hash function. This because it was designed to propagate ciphertext modifications to a trailing, predefined integrity control value, and not to produce good hash values from plaintext data.

EPBC conceals plaintext patterns by randomising the input of the block cipher with previous outputs of it. Similarly, ciphertext blocks result from the randomisation of the output of the block cipher with previous inputs of if. This double randomisation prevents attackers from gathering pairs of input and output blocks of the cipher in order to guess the cipher key. EPBC is similar to another error-propagating cipher mode, IOBC [8], but uses a different function in the plaintext feedback path. Such difference makes EPBC completely immune against known-plaintext attacks, thus more secure than IOBC, and makes it also faster than IOBC.

To assess the security qualities of EPBC, we show that EPBC guarantees confidentiality and integrity control of encoded data. Concerning confidentiality, we show that attackers cannot compute particular plaintext blocks even knowing all other plaintext and ciphertext blocks. Concerning integrity control, we show that attackers are unable to derive the correct tampering of the ciphertext produced by EPBC in order to perform a limited modification of the resulting plaintext. As a consequence, attackers can only try to tamper ciphertext blocks without any guaranties of success, and the probability of success is only given by the number of bits of the trailing integrity values used with EPBC.

Performance tests run on a SPARCstation 10/40 showed that, without considering the time expended by the block cipher, EPBC is 1.2 times faster than IOBC, in average, and 6.3 to 10.9 times faster than a combination of CBC and MD5.

The rest of the paper is structured as follows. The next section presents related work. Section 3 describes the algorithm of EPBC and how it guarantees confidentiality and integrity control of encoded data. In section 4 we evaluate the performance of EPBC. Finally, in section 5 we draw some conclusions.

2 Related Work

This section overviews some encryption modes providing error propagation and their weaknesses. These algorithms are the Block Chaining (BC [10]), the Cipher Block Chaining with Checksum (CBCC [7, 10]), the Propagating CBC (PCBC [6]), the PES_PCBC [14], and the Input and Output Block Chaining (IOBC [8]).

Hereafter C_i and P_i represent genuine ciphertext and plaintext blocks on the i-th iteration, c_i represents a tampered ciphertext block, p_i represents a plaintext block resulting from the decryption of a block from a tampered ciphertext, and $\mathbf{E}_K()$ and $\mathbf{D}_K()$ represent block encryption and decryption functions using key K.

Block Chaining (BC): The BC encryption mode uses all previous ciphertext blocks as feedback prior to encrypt a plaintext block [10].

$$\begin{aligned} \text{Encryption:} \quad & C_i = \mathbf{E}_K(P_i \oplus F_{i-1}) \\ \text{Decryption:} \quad & P_i = \mathbf{D}_K(C_i) \oplus F_{i-1} \\ \text{Encryption \& Decryption:} \quad & F_i = \bigoplus_{k=1}^{i} C_k \end{aligned}$$

The initial value of F_{i-1} is a secret initialisation vector. This encryption mode is very weak in detecting ciphertext tampering: all ciphertext blocks before the ones containing integrity control values can be shuffled, or pairs of ciphertext blocks can be XORed with an arbitrary value, without propagating erroneous decryptions to the remaining blocks.

Cipher Block Chaining with Checksum (CBCC): The CBCC encryption mode is a variant of CBC that keeps a XOR of all plaintext blocks, and XOR that with the last plaintext block before encryption; the last plaintext block is intended for integrity control using a constant value [7, 10]. This encryption mode is stronger than BC, but shuffling all ciphertext blocks, except the last one, does not affect the plaintext containing the integrity control value. The following example shows that we can recover the original value of the last plaintext block P_n by swapping C_1 with C_{n-1}:

$$\begin{cases} c_1 = C_{n-1} \\ c_{n-1} = C_1 \end{cases} \Longrightarrow \begin{cases} p_1 = P_{n-1} \oplus C_{n-2} \oplus IV \\ p_2 = P_2 \oplus C_1 \oplus C_{n-1} \\ p_{n-1} = P_1 \oplus C_{n-2} \oplus IV \\ p_n = P_n \end{cases}$$

Propagating CBC (PCBC): The PCBC encryption mode is similar to CBC and was used in the Kerberos Version 4 in order to simultaneously provide encryption and integrity control of data exchanged between Kerberos' principals and services [6, 12]. PCBC uses both plaintext and ciphertext feedback in order to propagate erroneous decryptions of a tampered encrypted message until the end of the message, rendering the entire message useless.

$$\begin{aligned} \text{Encryption:} \quad & C_i = \mathbf{E}_K(P_i \oplus P_{i-1} \oplus C_{i-1}) \\ \text{Decryption:} \quad & P_i = \mathbf{D}_K(C_i) \oplus P_{i-1} \oplus C_{i-1} \end{aligned}$$

The initial value of $P_{i-1} \oplus C_{i-1}$ is a secret initialisation vector. Like for CBCC, it is possible to shuffle encrypted blocks without propagating the erroneous decryption of those blocks until the last encrypted block; it only affects the corresponding plaintext blocks recovered after decryption and, possibly, the immediately following blocks [5]:

$$\begin{cases} c_i &= C_{i+1} \\ c_{i+1} &= C_{i+2} \\ c_{i+2} &= C_i \end{cases} \Longrightarrow \begin{cases} p_i &= P_{i-1} \oplus P_i \oplus P_{i+1} \oplus C_{i-1} \oplus C_i \\ p_{i+1} &= P_{i-1} \oplus P_i \oplus P_{i+2} \oplus C_{i-1} \oplus C_i \\ p_{i+2} &= P_{i+2} \oplus C_i \oplus C_{i+2} \\ p_{i+3} &= P_{i+3} \end{cases}$$

PES_PCBC: The PES_PCBC encryption mode was introduced in the Privacy Enhanced Sockets (PES) subsystem to simultaneously provide encryption and integrity control of data exchanged between client-server applications [14]. Like PCBC, PES_PCBC uses both plaintext and ciphertext feedback to achieve the error propagation effect.

$$\text{Encryption: } \begin{cases} C_i &= F_i \oplus G_{i-1} \\ F_i &= \mathbf{E}_K(G_i) \\ G_i &= P_i \oplus F_{i-1} \end{cases} \quad \text{Decryption: } \begin{cases} P_i &= F_{i-1} \oplus G_i \\ G_i &= \mathbf{D}_K(F_i) \\ F_i &= C_i \oplus G_{i-1} \end{cases}$$

The initial values of F_{i-1} and G_{i-1} are distinct, secret initialisation vectors.

The PES_PCBC encryption mode resists to attacks changing the order of ciphertext blocks but is weak against known-plaintext attacks. It is possible to compute tampered ciphertext blocks, resulting from the combination of plaintext and genuine ciphertext blocks, that defeat the desired error propagation effect. For example:

$$\begin{cases} c_i &= P_{i-1} \\ c_{i+1} &= P_i \oplus C_{i-1} \oplus C_{i+1} \\ c_{i+2} &= C_{i+2} \end{cases} \Longrightarrow \begin{cases} p_i &= C_{i-1} \\ p_{i+1} &= P_{i-1} \oplus P_{i+1} \oplus C_i \\ p_{i+2} &= P_{i+2} \end{cases}$$

Input and Output Block Chaining (IOBC): The IOBC encryption mode is similar to PES_PCBC but stronger concerning known-plaintext attacks [8]. Comparing with PES_PCBC, IOBC has an extra function in the plaintext feedback path that rotates feedback values before using them.

$$\text{Encryption: } \begin{cases} G_i &= P_i \oplus F_{i-1} \\ F_i &= \mathbf{E}_K(G_i) \\ C_i &= F_i \oplus \mathbf{f}(G_{i-1}) \end{cases} \quad \text{Decryption: } \begin{cases} F_i &= C_i \oplus \mathbf{f}(G_{i-1}) \\ G_i &= \mathbf{D}_K(F_i) \\ P_i &= G_i \oplus F_{i-1} \end{cases}$$

The initial values of F_{i-1} and G_{i-1} are distinct, secret initialisation vectors. The function $\mathbf{f}()$ makes two different rotations on the bits of G: one is applied to the most significative $b/2 - 1$ bits, and the other one is applied to the less significative $b/2 + 1$ bits; b is the number of bits of G and is assumed to be even.

The security of IOBC depends on the length of encrypted data; if it is longer than a given threshold length then it is subject to known-plaintext attacks. However, these attacks are much difficult to achieve than with PES_PCBC because

the attacker must know many plaintext blocks to perform it. For example, a tampered ciphertext block to start a known-plaintext attack is computed as follows:

$$\begin{cases} n = m \times (\frac{b}{2}+1) \times (\frac{b}{2}-1) = m \times (\frac{b^2}{4}-1) \quad \forall m \in \mathcal{N} \\ c_i = \mathbf{f}(P_{i-1}) \oplus \bigoplus_{k=1}^{n-1} \mathbf{f}^k\ (C_{i-2k}) \oplus \mathbf{f}^{k+1}\ (P_{i-2k-1}) \quad \text{for} \quad i, n \in \mathcal{N},\ \ i \geq 2n \end{cases}$$

For $m = 1$ and a typical value of $b = 64$, we get

$$n = 1023 \quad \Rightarrow \quad i \geq 2046$$

which means that attackers need to know $n = 1023$ specific plaintext blocks in order to start a known-plaintext attack. Therefore, for encrypted data shorter than 2046 64-bit blocks (≈ 16 Kbytes) it is impossible to start such an attack. For further details regarding the strength of IOBC against known-plaintext attacks see [8].

3 Efficient Error-Propagating Block Chaining (EPBC)

EPBC is a new block encryption mode resulting from an improvement of IOBC: it has a different function, $\mathbf{g}()$, in the plaintext feedback path (see Figure 1).

$$\text{Encryption:} \begin{cases} G_i = P_i \oplus F_{i-1} & (a) \\ F_i = \mathbf{E}_K(G_i) & \\ C_i = F_i \oplus \mathbf{g}(G_{i-1}) & (b) \end{cases} \tag{1}$$

$$\text{Decryption:} \begin{cases} F_i = C_i \oplus \mathbf{g}(G_{i-1}) & (a) \\ G_i = \mathbf{D}_K(F_i) & \\ P_i = G_i \oplus F_{i-1} & (b) \end{cases} \tag{2}$$

The initial values of F_{i-1} and G_{i-1} are distinct, secret initialisation vectors. Like PES_PCBC and IOBC, EPBC conceals plaintext patterns by randomising the input of the block cipher G_i with previous outputs of it (F_{i-1}). Similarly, ciphertext blocks C_i result from the randomisation of the output of the block cipher F_i with values derived from previous inputs of it ($\mathbf{g}(G_{i-1})$). This double

Fig. 1. Encryption/decryption of data blocks using the EPBC cipher mode and a block cipher. Dashed arrows represent value transfers at the end of each iteration, while solid arrows represent value transfers during each iteration.

randomisation prevents attackers from gathering (G_i, F_i) pairs in order to guess the encryption key K.

The function $\mathbf{g}()$ operates as follows, assuming that G values have an even number of bits, and that $G \equiv \langle G_H, G_L \rangle$, where G_H and G_L are the high and low order halves of G, respectively:

$$\mathbf{g}(G) = \langle G_H + \overline{G_L}, G_H \cdot \overline{G_L} \rangle \tag{3}$$

where the operators "+" and "·" represent the bitwise OR and AND operations, respectively, and $\overline{G_L}$ is the bitwise inverse of G_L. It is easy to show that the function $\mathbf{g}()$ is not injective, i.e. there are different arguments of $\mathbf{g}()$ that produce the same outcome (see Appendix A). As we will see below, that is an advantage because $\mathbf{g}()$ has no inverse.

Empirically, we found that for a domain with 2^b elements, each with b bits, the image of $\mathbf{g}()$ contains $3^{b/2}$ elements. For a typical value of $b = 64$, and a domain with 2^{64} elements, the image of $\mathbf{g}()$ includes $3^{32} \approx 2^{51}$ values. The fact that $\mathbf{g}()$ has a range smaller than its domain is not a security problem since such range is large enough for providing a good randomisation of F values.

The function $\mathbf{g}()$ is more efficient than the function $\mathbf{f}()$ of IOBC (see section 4), and has several properties that make it suitable for preventing known-plaintext attacks on EPBC. These properties, which are fully demonstrated in Appendix A, are the following:

$$\begin{cases} \forall x \in D_g & \mathbf{g}(x) \neq x & (a) \\ \nexists x \in D_g, \mathbf{h}() \quad \forall y \in D_g & \mathbf{g}(x \oplus y) = \mathbf{h}(x) \oplus y & (b) \\ \nexists \mathbf{h}() \quad \forall x, y \in D_g & \mathbf{g}(x \oplus y) = \mathbf{h}(x) \oplus \mathbf{h}(y) & (c) \end{cases} \tag{4}$$

where D_g is the domain of $\mathbf{g}()$. Expression (4a) is straightforward and needs no further explanation. Expression (4b) says that there is no x value in D_g and another function $\mathbf{h}()$ so that, for all y values in D_g, $\mathbf{g}(x \oplus y)$ is equal to $\mathbf{h}(x) \oplus y$. In other words, it means that even knowing the value of x, one cannot expand $\mathbf{g}(x \oplus y)$ in terms of another known value $\mathbf{h}(x)$ XORed with y. Expression (4c) says that there is no function $\mathbf{h}()$ so that, for all values of x and y in D_g, $\mathbf{g}(x \oplus y)$ is equal to $\mathbf{h}(x) \oplus \mathbf{h}(y)$.

Other similar functions, with an equal number of elements in the image and similar properties, could be used as $\mathbf{g}()$. These functions are:

$$\mathbf{g'}(G) = \langle \overline{G_H} + G_L, \overline{G_H} \cdot G_L \rangle$$
$$\mathbf{g''}(G) = \langle \overline{G_H} + \overline{G_L}, \overline{G_H} \cdot G_L \rangle$$

3.1 Confidentiality

Concerning confidentiality, it is necessary to prove that EPBC does not enable attackers to compute a specific plaintext block (P_i) even knowing all other plaintext and ciphertext blocks. Resolving equation (2b) to compute P_i we obtain:

$$
\begin{aligned}
P_i &= G_i \oplus F_{i-1} && (5)\\
&= G_i \oplus C_{i-1} \oplus \mathbf{g}\,(G_{i-2}) && (6)\\
&= G_i \oplus C_{i-1} \oplus \mathbf{g}\,(P_{i-2} \oplus F_{i-3})\\
&= G_i \oplus C_{i-1} \oplus \mathbf{g}\,(P_{i-2} \oplus C_{i-3} \oplus \mathbf{g}\,(G_{i-4})) && (7)\\
&= \cdots
\end{aligned}
$$

As the function $\mathbf{g}\,()$ is not injective, it has no inverse and the term G_i of equation (5) cannot be further expanded in terms of C_{i+1} and F_{i+1}. Since attackers ignore G values, and due to the characteristics of function $\mathbf{g}\,()$ given by expressions (4), they cannot compute or simplify, in order to isolate G values, expressions $\mathbf{g}\,(G_{i-2})$ in equation (6), and $\mathbf{g}\,(P_{i-2} \oplus C_{i-3} \oplus \mathbf{g}\,(G_{i-4}))$ in equation (7). Consequently, they cannot remove G terms from equations (6) and (7) and, thus, they cannot compute the value of the plaintext block P_i.

3.2 Integrity Control

Concerning integrity control, to tamper the ciphertext without affecting the decryption of trailing integrity control values attackers must perform a controlled modification of the ciphertext. This means that attackers must provide a correct sequence of false ciphertext blocks (c values) in order to modify and latter recover the correct internal state (G and F values) of the decryption engine before the actual decryption of the integrity control values. This implies that all f values resulting from c values during decryption must be correct F values. Otherwise, f values would decrypt to something unpredictable, instead of correct G values, and attackers would loose control of the tampering. Therefore, the possible values of a c_i block are:

$$
\begin{aligned}
c_i &= F_j \oplus \mathbf{g}\,(G_{i-1}) \quad \wedge \quad i \neq j\\
&= F_j \oplus \mathbf{g}\,(P_{i-1} \oplus F_{i-2}) && (8)\\
&= C_j \oplus \mathbf{g}\,(G_{j-1}) \oplus \mathbf{g}\,(P_{i-1} \oplus F_{i-2})\\
&= C_j \oplus \mathbf{g}\,(P_{j-1} \oplus F_{j-2}) \oplus \mathbf{g}\,(P_{i-1} \oplus F_{i-2}) && (9)\\
&= \cdots
\end{aligned}
$$

We could further expand equation (9) but it would not simplify the task of attackers. Since attackers ignore F values, and due to the characteristics of function $\mathbf{g}\,()$ given by expressions (4), they cannot compute or simplify, in order to isolate F values, expressions $\mathbf{g}\,(P_{i-1} \oplus F_{i-2})$ in equations (8) and (9), and $\mathbf{g}\,(P_{j-1} \oplus F_{j-2})$ in equation (9). Consequently, they cannot remove F terms from equations (8) and (9) and, thus, they cannot compute a false ciphertext block c_i in order to start a controlled modification of the encrypted data.

In conclusion, the EPBC encryption mode does not suffer from the weaknesses of all other modes presented in section 2. Attackers are unable to take

control of the EPBC decryption engine by providing tampered ciphertext blocks. Therefore, they can only try to tamper ciphertext blocks without any guaranties of success, and the probability of success is only given by the number of bits of the trailing integrity values used with EPBC.

4 Performance Evaluation

In this section we evaluate the performance of EPBC, and we compare it with other techniques used to achieve confidentiality and integrity control: the IOBC error-propagating encryption mode and a common combination of an encryption mode (CBC) and a one-way hash function (MD5). The performance tests were executed on a Sun SPARCstation 10/40.

In order to do a fair comparison between different encryption modes, we used our own optimised implementations. Our implementations do not modify source bytes, either plaintext or ciphertext, and were optimised in order to reduce the number of function calls and maximise the use of CPU registers instead of memory accesses. Encryption/decryption cycles were unrolled 4 times. In Appendix B we present our implementation of CBC, IOBC, and EPBC.

Table 1 presents the average elapsed time per block expended by CBC, IOBC and EPBC when encoding and decoding arrays of 64-bit blocks with two different lengths: 128 blocks and 1 M blocks. These two different lengths allow us to assess the impact of the memory cache status, either warm or cold, in the performance of the algorithms. The time measurements do not include the time expended by the block cipher function and were obtained by dividing the total elapsed time expended in encoding and decoding a total of 160 M blocks (1280 Mbytes). These values show that, on average, the EPBC encryption mode is 1.2 times faster than IOBC and 1.5 times slower than CBC.

Cache status	Algorithm	Encryption (ns)	Decryption (ns)
Warm	CBC	166	177
	IOBC	344	369
	EPBC	268	294
	CBC & MD5	2 910	2 918
Cold	CBC	334	337
	IOBC	519	529
	EPBC	455	458
	CBC & MD5	2 894	2 879

Table 1. Average user time per block expended on a Sun SPARCstation 10/40 by three encryption modes (CBC, IOBC and EPBC) and a combination of an encryption mode and a one-way hash function (CBC & MD5), when encoding and decoding arrays of 64-bit blocks (not including the time expended by the block cipher function).

We did the same performance evaluations for a combination of the CBC encryption mode and the MD5 one-way hash function, using our implementation of CBC and the MD5 implementation presented in the document describing it [9]. As input for MD5 we used all the blocks of the arrays except the last two; these were used to store the resulting hash value. The average user time expended in processing each block with CBC and MD5 is also presented in Table 1.

The values in Table 1 show that the combination of CBC and MD5 is almost insensible to the cache status. The small speedup that is observed in the measurements obtained in the simulations with the cold cache is due to the fact that we execute less function calls to encode/decode all the 160 M blocks. Comparing with EPBC, we can see that this is 6.3 to 10.9 times faster than the combination of CBC and MD5. If we had used SHA [11] instead of MD5, the performance gain would probably be greater. According to a table presented by Schneier [10] for a 486 SX processor, the SHA function is about two times slower than MD5.

5 Conclusions

In this document we presented a new encryption mode, Efficient Error-Propagating Block Chaining, that propagates erroneous decryptions of tampered ciphertext blocks to all following blocks. This encryption mode allows integrity validation of the recovered plaintext by checking a predefined value at the end of the plaintext. This value may be set up in many different ways: agreed between interacting peers, derived from a secret value, like the encryption key, or efficiently computed from some bits, but not necessarily all, of the plaintext data.

Unlike other encryption modes also providing error propagation, such as BC, CBCC, PCBC, PES_PCBC, or IOBC, the EPBC encryption mode is not vulnerable to attacks changing the order of ciphertext blocks or known-plaintext attacks. It also conceals plaintext patterns by randomising the input of the block cipher with previous outputs of it, and prevents attackers from gathering pairs of cipher input and output blocks in order to guess the encryption key.

Performance tests run on a SPARCstation 10/40 showed that, without considering the time expended by the block cipher, EPBC is in average 1.2 times faster than IOBC, and 6.3 to 10.9 times faster than a combination of CBC and MD5. The performance gain of EPBC when compared with the combination of CBC and MD5 can be very important if we are concerned with providing together data confidentiality and integrity control. This is the case of most secure communication protocols, such as SSL [3], PES [14], or SKIP [1], where interacting applications or operating systems can use constant or easy-to-compute, secret integrity control values for one or more secure communication channels. This way they could save a significative amount of processing time when exchanging large quantities of confidential information through secure communication channels.

References

1. Ashar Aziz, Tom Markson, and Hemma Prafullchandra. Simple Key-Management For Internet Protocols (SKIP). Internet Draft, Sun Microsystems, Inc., December 1995.
2. D. Balenson. Privacy Enhancement for Internet Electronic Mail (Part III): Algorithms, Modes, and Identifiers. RFC 1423, IAB IRTF PSRG, IETF PEM WG, February 1993.
3. Alan O. Freier, Philip Karlton, and Paul C. Kocher. SSL Protocol Version 3.0. Internet Draft, Netscape Communications Corp., March 1996.
4. Information Processing - Modes of Operation for an n-bit Block Cipher Algorithm. ISO IEC/DIS 10116, 1989.
5. J. T. Kohl. The Use of Encryption in Kerberos for Network Authentication. In *Advances in Cryptology – CRYPTO '89 Proceedings*, pages 35–43. Springer-Verlag, 1990.
6. C. H. Meyer and S. M. Matyas. *Cryptography: A New Dimension in Computer Data Security.* John Wiley & Sons, Inc., New York, 1982.
7. Xerox Network System (XNS) Authentication Protocol. XSIS 098404, Xerox Corporation, April 1984.
8. Francisco Recacha. IOBC: Un nuevo modo de encadenamiento para cifrado en bloque. In *Proc. of the IV Reunion Espanyola sobre Criptologia*, Valladolid, September 1996.
9. R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, MIT Laboratory for Computer Science and RSA Data Security, Inc., April 1992.
10. Bruce Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C.* John Wiley & Sons, Inc., second edition, 1996.
11. Secure Hash Standard. NIST FIPS PUB 180, April 1993.
12. Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In *Proc. of the USENIX Winter Conf.*, pages 191–202, Dallas, Texas, USA, February 1988.
13. Philip Zimmermann. *The Official PGP User's Guide.* MIT Press, 1995.
14. André Zúquete and Paulo Guedes. Transparent Authentication and Confidentiality for Stream Sockets. *IEEE Micro*, 16(3):34–41, June 1996.

A Demonstrations

In this Appendix we will demonstrate some of the properties of the function $\mathbf{g}()$ previously introduced in section 3. As a reminder, the function $\mathbf{g}()$ operates as follows:

$$\mathbf{g}(x) = \langle x_H + \overline{x_L}, x_H \cdot \overline{x_L} \rangle \tag{10}$$

where x is a value with an even number of bits (b), and $x \equiv \langle x_H, x_L \rangle$, where x_H and x_L are the high and low order halves of x, respectively. The operators "+" and "·" represent the bitwise OR and AND operations, respectively, and $\overline{x_L}$ is the bitwise inverse of x_L.

1^{st} property: $\mathbf{g}()$ is not injective

Demonstration: The function $\mathbf{g}()$ is not injective if:

$$\exists x, y \in D_g, x \neq y \quad \mathbf{g}(x) = \mathbf{g}(y)$$

Expanding $\mathbf{g}(x) = \mathbf{g}(y)$ using equation (10), we get:

$$\mathbf{g}(x) = \mathbf{g}(y) \iff \begin{cases} x_H + \overline{x_L} = y_H + \overline{y_L} \\ x_H \cdot \overline{x_L} \ = y_H \cdot \overline{y_L} \end{cases}$$

This two equations can be resolved in several ways. One solution, for example, is $y_H = \overline{x_L}$ and $y_L = \overline{x_H}$. Therefore, $\mathbf{g}()$ is not injective.

2^{nd} property: $\forall x \in D_g \quad \mathbf{g}(x) \neq x$

Demonstration: If the opposite is true, then:

$$\exists x \in D_g \quad \mathbf{g}(x) = x \iff \begin{cases} x_H + \overline{x_L} = x_H \\ x_H \cdot \overline{x_L} \ = x_L \end{cases}$$

Expanding these two equations in terms of the individual bits of each of the halves of x, we have:

$$\forall i \in \{1, 2, \cdots, b/2\} \ \begin{cases} x_{Hi} + \overline{x_{Li}} = x_{Hi} \Leftrightarrow x_{Li} = 1 \vee x_{Hi} = 1 \\ x_{Hi} \cdot \overline{x_{Li}} \ = x_{Li} \Leftrightarrow x_{Li} = 0 \wedge x_{Hi} = 0 \end{cases}$$

As there are no suitable values for the pair of bits (x_{Li}, x_{Hi}), the opposite is false and, thus, the property holds.

3^{th} property: $\nexists x \in D_g, \mathbf{h}() \quad \forall y \in D_g \quad \mathbf{g}(x \oplus y) = \mathbf{h}(x) \oplus y$

Demonstration: If the opposite is true, then:

$$\exists x \in D_g, \mathbf{h}()$$

$$\mathbf{g}(x \oplus y) = \mathbf{h}(x) \oplus y \iff \begin{cases} (x_H \oplus y_H) + \overline{(x_L \oplus y_L)} = \mathbf{h}(x)_H \oplus y_H \\ (x_H \oplus y_H) \cdot \overline{(x_L \oplus y_L)} \ = \mathbf{h}(x)_L \oplus y_L \end{cases}$$

As the first equation depends on y_L and the second equation depends on y_H, that implies that $\mathbf{h}(x)$ depends on y, which cannot not happen. Therefore, the property holds.

4^{th} property: $\nexists \mathbf{h}() \quad \forall x, y \in D_g \quad \mathbf{g}(x \oplus y) = \mathbf{h}(x) \oplus \mathbf{h}(y)$

Demonstration: If we have $x = y$, then:

$$\begin{cases} \mathbf{g}(x \oplus y) = \mathbf{g}(0) \neq 0 \\ \mathbf{h}(x) \oplus \mathbf{h}(y) = 0 \end{cases} \implies$$

$$\implies \forall x, y \in D_g, x = y \quad \forall \mathbf{h}() \quad \mathbf{g}(x \oplus y) \neq \mathbf{h}(x) \oplus \mathbf{h}(y)$$

Consequently, the property holds.

B Source Code of CBC, IOBC and EPBC

```
/*  Global macros  */
#define w32             unsigned int    /* 32 bit word */
#define CIPHER(to,from)                 /* nothing */
#define DECIPHER(to,from)               /* nothing */
#define DO_4_TIMES(op)  {op op op op}
```

```
#define CBC_ENC_BLOCK \
  I[0] = *P++ ^ c0; I[1] = *P++ ^ c1;      \
  CIPHER(C,I);                             \
  c0 = *C++; c1 = *C++;

cbcEnc ( int n, w32 *C, w32 *P, w32 *C_1 )
{
  register w32 c0 = C_1[0], c1 = C_1[1];
  w32 I[2];

  for (; n >= 4; n -= 4)
    DO_4_TIMES( CBC_ENC_BLOCK )
  while (n--) { CBC_ENC_BLOCK }
}
```

```
#define CBC_DEC_BLOCK \
  DECIPHER(P,C);                           \
  *P++ ^= c0; *P++ ^= c1;                  \
  c0 = *C++; c1 = *C++;

cbcDec ( int n, w32 *C, w32 *P, w32 *C_1 )
{
  register w32 c0 = C_1[0], c1 = C_1[1];

  for (; n >= 4; n -= 4)
    DO_4_TIMES( CBC_DEC_BLOCK )
  while (n--) { CBC_DEC_BLOCK }
}
```

```
#define ROT_L(h,l) (((l & 0xFFFFFFFC) >> 1) | ((l & 2) << 30) | (h & 1))
#define ROT_H(h,l) ((h >> 1) | (l << 31))
```

```
#define IOBC_ENC_BLOCK \
  c0 = ROT_L(g1,g0); c1 = ROT_H(g1,g0);    \
  G[0] = g0 = *P++ ^ f0;                   \
  G[1] = g1 = *P++ ^ f1;                   \
  CIPHER(F,G);                             \
  f0 = F[0]; f1 = F[1];                    \
  *C++ = c0 ^ f0; *C++ = c1 ^ f1;

iobcEnc ( int n, w32 *C, w32 *P,
                 w32 *F, w32 *G )
{
  register w32 f0 = F[0], f1 = F[1];
  register w32 g0 = G[0], g1 = G[1];
  register w32 c0, c1;

  for (; n >= 4; n -= 4)
    DO_4_TIMES( IOBC_ENC_BLOCK )
  while (n--) { IOBC_ENC_BLOCK }
}
```

```
#define IOBC_DEC_BLOCK \
  p0 = f0; p1 = f1;                        \
  F[0] = f0 = *C++ ^ ROT_L(g1,g0);         \
  F[1] = f1 = *C++ ^ ROT_H(g1,g0);         \
  DECIPHER(G,F);                           \
  g0 = G[0]; g1 = G[1];                    \
  *P++ = p0 ^ g0; *P++ = p1 ^ g1;

iobcDec ( int n, w32 *C, w32 *P,
                 w32 *F, w32 *G )
{
  register w32 f0 = F[0], f1 = F[1];
  register w32 g0 = G[0], g1 = G[1];
  register w32 p0, p1;

  for (; n >= 4; n -= 4)
    DO_4_TIMES( IOBC_DEC_BLOCK )
  while (n--) { IOBC_DEC_BLOCK }
}
```

```
#define EPBC_ENC_BLOCK \
  c0 = ~g0 & g1; c1 = ~g0 | g1;            \
  G[0] = g0 = *P++ ^ f0;                   \
  G[1] = g1 = *P++ ^ f1;                   \
  CIPHER(F,G);                             \
  f0 = F[0]; f1 = F[1];                    \
  *C++ = c0 ^ f0; *C++ = c1 ^ f1;

epbcEnc ( int n, w32 *C, w32 *P,
                 w32 *F, w32 *G )
{
  register w32 f0 = F[0], f1 = F[1];
  register w32 g0 = G[0], g1 = G[1];
  register w32 c0, c1;

  for (; n >= 4; n -= 4)
    DO_4_TIMES( EPBC_ENC_BLOCK )
  while (n--) { EPBC_ENC_BLOCK }
}
```



```
#define EPBC_DEC_BLOCK \
  p0 = f0; p1 = f1;                        \
  F[0] = f0 = *C++ ^ (~g0 & g1);           \
  F[1] = f1 = *C++ ^ (~g0 | g1);           \
  DECIPHER(G,F);                           \
  g0 = G[0]; g1 = G[1];                    \
  *P++ = p0 ^ g0; *P++ = p1 ^ g1;

epbcDec ( int n, w32 *C, w32 *P,
                 w32 *F, w32 *G )
{
  register w32 f0 = F[0], f1 = F[1];
  register w32 g0 = G[0], g1 = G[1];
  register w32 p0, p1;

  for (; n >= 4; n -= 4)
    DO_4_TIMES( EPBC_DEC_BLOCK )
  while (n--) { EPBC_DEC_BLOCK }
}
```