The Extended Codebook (XCB) Mode of Operation Version 3

David A. McGrew and Scott R. Fluhrer
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95032
{mcgrew,sfluhrer}@cisco.com

August 2, 2007

Abstract

We describe a block cipher mode of operation that implements a 'tweakable' (super) pseudorandom permutation with an arbitrary block length and arbitrary tweak length. This mode can be used to provide the best possible security in systems that cannot allow data expansion, such as disk-block encryption and some network protocols. The mode accepts an additional input ('tweak') that can be used to protect against attacks that manipulate the ciphertext by rearranging the ciphertext blocks. We call this mode the Extended Codebook (XCB). This document describes version three of XCB; it refines an earlier version that appeared in 2004.

XCB is similar to a five-round Luby-Rackoff cipher in which the first and last rounds do not use the conventional Feistel structure, but instead use a single block cipher invocation. The third round is an unbalanced Feistel structure that uses counter mode as a pseudorandom function. The second and fourth rounds are unbalanced Feistel structures using a universal hash function; we re-use the polynomial hash over a binary field defined in the Galois/Counter Mode (GCM) of operation for block ciphers. This choice provides efficiency in both hardware and software and allows for implementation re-use. XCB also has several other useful properties. It is highly efficient, and it accepts arbitrarily-sized plaintexts and associated data, including any plaintexts with lengths that are no smaller than the width of the block cipher (as long as the additional input contains a nonce).

Contents

1 Introduction

There are several scenarios in which *length-preserving, deterministic encryption* is useful. An encryption method is length-preserving if the ciphertext has exactly the same number of bits as does the plaintext. Such a method must be deterministic, since it is impossible to accommodate random data (such as an initialization vector) within the ciphertext. In some cases, deterministic length-preserving encryption exactly matches the requirements. For example, in some encrypted database applications, determinism is essential in order to ensure a direct correspondence between plaintext values being looked up and previously stored ciphertext values.

In some other cases, there is a length-preservation requirement that makes it impossible to provide all of the security services that are desired. Length-preserving algorithms cannot provide message authentication, since there is no room for a message authentication code, and they cannot meet some strong definitions of confidentiality [**?**]. Essentially, these algorithms implement a codebook; repeated encryptions of the same plaintext value with the same key result it identical ciphertext values. An adversary gains knowledge about the plaintext by seeing which ciphertext values match, and which do not match. Despite these limitations, in many scenarios it may be desirable to use length-preserving encryption because other methods are unworkable. Length-preservation may allow encryption to be introduced into data processing systems that have already been implemented and deployed. Many network protocols have fixed-width fields, and many network systems have hard limits on the amount of data expansion that is possible. One important example is that of disk-block encryption, which is currently being addressed in the IEEE Security in Storage Working Group [**?**].

Given the limitation of length-preservation, the best security that we can provide is *non-malleable* encryption. Informally, a cipher is non-malleable if changing a single bit of a ciphertext value affects all of the bits of the corresponding plaintext. More formally, we require our cipher to be a *pseudorandom permutation*; it is indistinguishable from a permutation on the set of messages to a computationally bounded adversary. Because we want our cipher to handle plaintexts whose size may vary, we require the cipher to be a pseudorandom *arbitrary length* permutation: for each of the possible plaintext lengths, the cipher acts as a pseudorandom permutation. To provide as much flexibility as possible, we allow the plaintext lengths to vary even for a single fixed key.

In some cases, some additional data can be associated with the plaintext. By using this data as an input, we can provide better security, by letting each distinct associated data value 'index' a pseudorandom permutation. That is, we require the cipher to be a *pseudorandom arbitrary-length permutation with associated data*: for each plaintext length and each value of the associated data, the cipher acts as a pseudorandom permutation. For maximum flexibility, we allow the length of the associated data field to vary even for a single fixed key. In the disk block example, we can use the block number as the associated data value. This will prevent some attacks which rely on the codebook property, since identical plaintext values encrypted with distinct associated data values give unrelated ciphertext values.

The use of an associated data input to a pseudorandom permutation first appeared in the innovative Hasty Pudding Cipher of Schroppel [**?**], where it was called a 'spice', and was given a rigorous mathematical treatment by Liskov, Rivest, and Wagner [**?**], who called it a 'tweak'. Our security

goal follows that of the latter work, with the distinction that we allow the associated data to have an arbitrary length.

The extended codebook (XCB) mode of operation for block ciphers implements a pseudorandom arbitrary-length permutation with associated data. This mode is amenable to implementation in both hardware and software, and it has a computational cost that is relatively low (compared to other modes that meet the same security goals). It requires only $n + 1$ block cipher invocations and $2n + 6$ multiplications in $GF(2^w)$, where w is the number of bits in the block cipher inputs and outputs. The mode also has several useful properties: it accepts arbitrarily-sized plaintexts and associated data, including any plaintexts with lengths of at least w bits, when the associated data contains a nonce. This property allows XCB to protect short data, like the common 20-byte G.729 voice codec in Secure RTP [**?**]. The basic components of XCB are identical to those of the Galois/Counter Mode (GCM) of operation, making XCB easy to implement given an implementation of GCM, and making compact GCM/XCB implementations possible.

In the following, we review the changes from the first version (Section 1.1), define XCB (Section 2) and provide a security rationale for its design (Section **??**).

1.1 Version Information

The initial version of XCB appeared on the IACR eprint website in 2004 [**?**]. A revised version followed in 2005 [**?**]; that version improved on the earlier one by being more amenable to software implementation; it reduces the per-key storage requirement by over 50%.

This is the third version of XCB. It incorporates changes that make security properties easier to analyze, and it provides greater robustness. Additionally, it reorders the operations in a way that makes XCB more amenable to pipelined implementation.

2 Specification

This section contains the complete normative specification for XCB for use with 128-bit block ciphers. In order to use XCB with other block cipher widths, it would be necessary to define a representation for the finite field of the appropriate size.

2.1 Interface

The encryption operation takes as input a secret key K, a plaintext P, and associated data Z, and outputs a ciphertext C. This operation is denoted as $C = E(K, Z, P)$. The values K, P, Z, and C are bit strings. The length of C is identical to that of P.

The decryption operation takes as input a secret key K, a ciphertext C, and associated data

Z, and outputs a plaintext P. This operation is denoted as $P = D(K, Z, C)$. The identity $D(K, Z, E(K, Z, P)) = P$ holds for all values of K and Z.

There are two distinct ways in which XCB can be used. For any fixed value of the key, if all of the values of the associated data Z in all of the encryption operations are distinct, then the plaintext can have a length between w and 2^{39} bits, inclusive. We call this *nonce mode*. Otherwise, if the associated data values are *not* distinct, then the plaintext must have a length between $2w$ and 2^{39} bits, inclusive. We call this *normal mode*.

2.2 Notation

The two main functions used in XCB are block cipher encryption and multiplication over the field $GF(2^{128})$. The block cipher encryption of the value X with the key K is denoted as $\mathbf{e}(K, X)$, and the block cipher decryption is denoted as $\mathbf{d}(K, X)$. (Note that we reserve the symbols E and D to denote XCB encryption and decryption, respectively.) The number of bits in the inputs and outputs of the block cipher is denoted as w. For the Advanced Encryption Standard (AES), $w = 128$. The multiplication of two elements $X, Y \in GF(2^{128})$ is denoted as $X \cdot Y$, and the addition of X and Y is denoted as $X \oplus Y$. Addition in this field is equivalent to the bitwise exclusive-or operation, and the multiplication operation is defined in Section 2.4. We denote the number of bits in a bit string X as $\#X$.

The function $\mathrm{len}(S)$ returns a $w/2$-bit string containing the nonnegative integer describing the number of bits in its argument S, with the least significant bit on the right. The expression 0^l denotes a string of l zero bits, and $A \| B$ denotes the concatenation of two bit strings A and B. The function $\mathbf{msb}_t(S)$ returns the initial t bits of the string S. We consider bit strings to be indexed starting on the left, so that bit zero of S is the leftmost bit. When S is a bit string and $0 \le a < b < \#S$, we denote as $S[a; b]$ the length $b - a$ substring of S consisting of bits a through b of S. The symbol $\{\}$ denotes the bit string with zero length.

2.3 Definition

The XCB encryption and decryption operations are defined in Algorithms 1 and 2, respectively, and the encryption operation is illustrated in Figure 1. These algorithms use the block cipher encryption functions $\mathbf{e}$ and $\mathbf{d}$, as well as the hash function $\mathbf{h}$ and the pseudorandom function $\mathbf{c}$. The variables H, K_e, K_d and K_c are derived from K, essentially by running the function $\mathbf{e}$ in counter mode.

Optionally, these values can be stored between evaluations of these algorithms, in order to trade off some storage for a decreased computational load.

The function $\mathbf{c} : \{0,1\}^k \times \{0,1\}^w \to \{0,1\}^l$, where the output length l is is bounded by $0 \le l \le 2^{39}$, generates an arbitrary-length output by running the block cipher $\mathbf{e}$ in counter mode, using its w-bit

Algorithm 1 The XCB encryption operation. Given a key $K \in \{0,1\}^k$, a plaintext $\mathbf{P} \in \{0,1\}^m$ where $m \in [w, 2^{39}]$, and associated data $Z \in \{0,1\}^n$ where $n \in [0, 2^{39}]$, this operation returns a ciphertext $\mathbf{C} \in \{0,1\}^m$.

$H \leftarrow \mathbf{e}(K, 0^w)$
$K_e \leftarrow \mathbf{msb}_k(\mathbf{e}(K, 0^{w-3}\|001)\|\mathbf{e}(K, 0^{w-3}\|010))$
$K_d \leftarrow \mathbf{msb}_k(\mathbf{e}(K, 0^{w-3}\|011)\|\mathbf{e}(K, 0^{w-3}\|100))$
$K_c \leftarrow \mathbf{msb}_k(\mathbf{e}(K, 0^{w-3}\|101)\|\mathbf{e}(K, 0^{w-3}\|110))$
$A \leftarrow \mathbf{P}[\#\mathbf{P} - w; \#\mathbf{P} - 1]$
$B \leftarrow \mathbf{P}[0; \#\mathbf{P} - w - 1]$
$C \leftarrow \mathbf{e}(K_e, A)$
$D \leftarrow C \oplus \mathbf{h}_1(H, Z, B)$
$E \leftarrow B \oplus \mathbf{c}(K_c, D, \#B)$
$F \leftarrow D \oplus \mathbf{h}_2(H, Z, E)$
$G \leftarrow \mathbf{d}(K_d, F)$
return $E\|G$

Algorithm 2 The XCB decryption operation. Given a key $K \in \{0,1\}^k$, a ciphertext $\mathbf{C} \in \{0,1\}^m$ where $m \in [w, 2^{39}]$, and associated data $Z \in \{0,1\}^n$ where $n \in [0, 2^{39}]$, returns a plaintext $\mathbf{P} \in \{0,1\}^m$.

$H \leftarrow \mathbf{e}(K, 0^w)$
$K_e \leftarrow \mathbf{msb}_k(\mathbf{e}(K, 0^{w-3}\|001)\|\mathbf{e}(K, 0^{w-3}\|010))$
$K_d \leftarrow \mathbf{msb}_k(\mathbf{e}(K, 0^{w-3}\|011)\|\mathbf{e}(K, 0^{w-3}\|100))$
$K_c \leftarrow \mathbf{msb}_k(\mathbf{e}(K, 0^{w-3}\|101)\|\mathbf{e}(K, 0^{w-3}\|110))$
$G \leftarrow \mathbf{C}[\#\mathbf{C} - w; \#\mathbf{C} - 1]$
$E \leftarrow \mathbf{C}[0; \#\mathbf{C} - w - 1]$
$F \leftarrow \mathbf{e}(K_e, G)$
$D \leftarrow F \oplus \mathbf{h}_2(H, Z, E)$
$B \leftarrow E \oplus \mathbf{c}(K_c, D, \#E)$
$C \leftarrow D \oplus \mathbf{h}_1(H, Z, B)$
$A \leftarrow \mathbf{d}(K_d, C)$
return $B\|A$

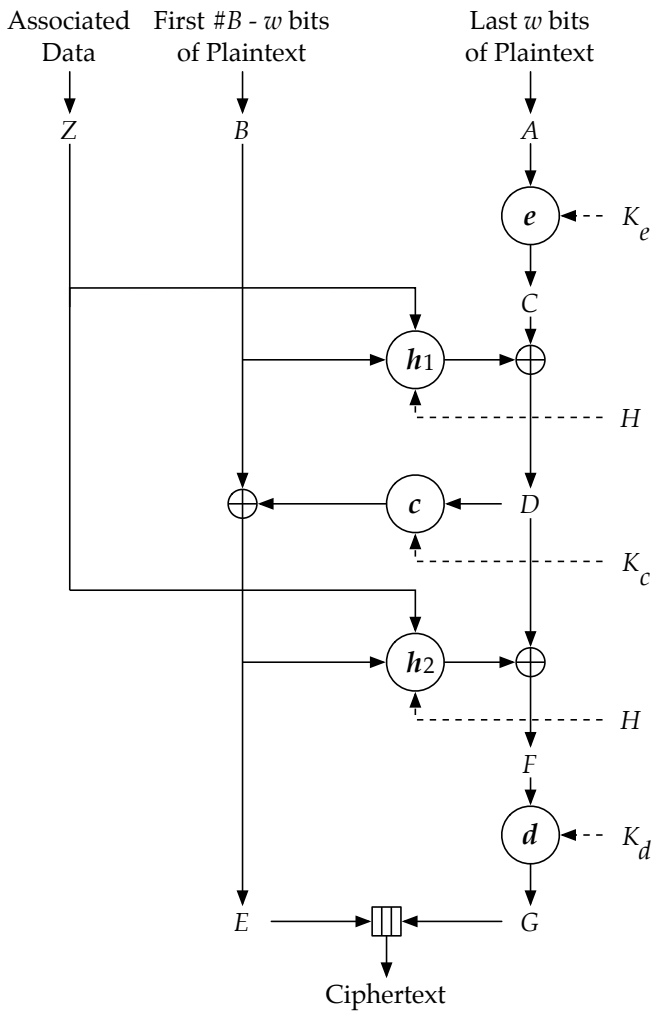

Figure 1: The XCB encryption operation. The secret variables H, K_e, K_d, and K_c are derived from the secret key K.

input as the initial counter value. Its definition is

$$\mathbf{c}(K, W, l) = E(K, W) \| E(K, \mathrm{incr}(W) \| \ldots \| \mathbf{msb}_t(E(K, \mathrm{incr}^{n-1}(W)), \tag{1}$$

where we make the output length l an explicit parameter for clarity; $n = \lceil l/w \rceil$ is the number of w-bit blocks in the output and $t = l \bmod w$ is number of bits in the trailing block. Here the function incr : $\{0,1\}^w \to \{0,1\}^w$ is the increment operation that is used to generate successive counter values. This function treats the rightmost 32 bits of its argument as a nonnegative integer with the least significant bit on the right, increments this value modulo 2^{32}. More formally,

$$\mathrm{incr}(X) = X[0; w-33] \,\|\, (X[w-32; w-1] + 1 \bmod 2^{32}), \tag{2}$$

where we rely on the implicit conversion of bit strings to integers.

The functions $\mathbf{h}_1$ and $\mathbf{h}_2$ are defined in terms of the underlying hash function $\mathbf{h}$ as

$$\begin{aligned}\mathbf{h}_1(H, Z, B) &= \mathbf{h}(H, 0^w \| Z, B \| 0^{\#B \bmod w + w}) \\ \mathbf{h}_2(H, Z, B) &= \mathbf{h}(H, Z \| 0^w, E \| 0^{\#B \bmod w} \| \text{len}(Z \| L) \| \text{len}(B))\end{aligned} \tag{3}$$

The function $\mathbf{h} : \{0,1\}^w \times \{0,1\}^m \times \{0,1\}^n \rightarrow \{0,1\}^w, m \in [w, 2^{39}], n \in [0, 2^{39}]$ is defined by $\mathbf{h}(H, A, C) = X_{m+n+1}$, where the variables $X_i \in \{0,1\}^w$ for $i = 0, \ldots, m+n+1$ are defined as

$$X_i = \begin{cases} 0 & \text{for } i = 0 \\ (X_{i-1} \oplus A_i) \cdot H & \text{for } i = 1, \ldots, m-1 \\ (X_{m-1} \oplus (A_m^* \| 0^{w-v})) \cdot H & \text{for } i = m \\ (X_{i-1} \oplus C_{i-m}) \cdot H & \text{for } i = m+1, \ldots, m+n-1 \\ (X_{m+n-1} \oplus (C_n^* \| 0^{w-u})) \cdot H & \text{for } i = m+n \\ (X_{m+n} \oplus (\text{len}(A) \| \text{len}(C))) \cdot H & \text{for } i = m+n+1. \end{cases} \tag{4}$$

Here we let A_i denote the w-bit substring $A[(i-1)w; iw-1]$, and let C_i denote $C[(i-1)w; iw-i]$. In other words, A_i and C_i are the i^{th} blocks of A and C, respectively, if those bit strings are decomposed into w-bit blocks. This function is identical to GHASH, the universal hash that is used as a component of the Galois/Counter Mode (GCM) of Operation [**?**], except that GHASH requires $w = 128$, as is the case for AES [**?**].

2.4 Multiplication in $GF(2^{128})$

The multiplication operation is defined as an operation on bit vectors in order to simplify the specification; it allows us to keep finite field mathematics out of the normative definition of the algorithm. Background information on this field and its representation, and strategies for efficient implementation, is provided in the GCM specification [**?**, Sections 3 and 4]. This definition of multiplication corresponds to the polynomial basis with the field polynomial of $f = 1 + \alpha + \alpha^2 + \alpha^7 + \alpha^{128}$.

Each field element is a vector of 128 bits. The i^{th} bit of an element X is denoted as X_i. The leftmost bit is X_0, and the rightmost bit is X_{127}. The multiplication operation uses the special element $R = 11100001 \| 0^{120}$, and is defined in Algorithm 3. The function rightshift() moves the bits of its argument one bit to the right. More formally, whenever $W = \text{rightshift}(V)$, then $W_i = V_{i-1}$ for $1 \leq i \leq 127$ and $W_0 = 0$.

Algorithm 3 Multiplication in $GF(2^{128})$. Returns $Z = X \cdot Y$, where $X, Y, Z \in GF(2^{128})$.

$Z \leftarrow 0$, $V \leftarrow X$
for $i = 0$ to 127 **do**
  if $Y_i = 1$ **then**
    $Z \leftarrow Z \oplus V$
  end if
  if $V_{127} = 0$ **then**
    $V \leftarrow \text{rightshift}(V)$
  else
    $V \leftarrow \text{rightshift}(V) \oplus R$
  end if
end for
return Z