AN ABSTRACT OF THE THESIS OF

Lelia C. Barlow for the degree of
Master of Science in
Electrical and Computer Engineering presented
on May 19, 2005.
Title: Symmetric Encryption with Multiple Keys: Techniques and Applications.

Abstract approved:

_____

Çetin K. Koç

Symmetric key block ciphers, such as AES, are well known and frequently used. There are five confidentiality modes of operation that are currently recommended for use with a symmetric key block cipher algorithm: Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and Counter (CTR).

This paper investigates extending the block cipher modes of operation using multiple symmetric keys. Unlike traditional multiple-key encryption techniques, each block of plaintext is encrypted using a single key. The key used to encrypt a block is selected by the corresponding entry of a sequence. Each sequence entry represents one of the independent keys in the set.

This paper also explores a variety of applications of this technique. Possible applications include establishing a type of covert channel, customizing message encryption at the block level, and expanding a string of pseudo-random numbers. Simulations have been constructed to illustrate the concepts.

Symmetric Encryption with Multiple Keys: Techniques and Applications

by
Lelia C. Barlow

A THESIS
submitted to
Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented May 19, 2005
Commencement June 2005

Master of Science thesis of Lelia C. Barlow
presented on May 19, 2005.

APPROVED:

_____

Major Professor, representing Electrical and Computer Engineering

_____

Director of the School of Electrical Engineering and Computer Science

_____

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

_____

Lelia C. Barlow, Author

ACKNOWLEDGEMENTS

I would like to thank my major advisor, Dr. Koç. I would also like to thank my other committee members, Dr. Bose, Dr. Schmidt, and Dr. Pohjanpelto.

I sincerely appreciate the support and encouragement from my family and friends.

TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

LIST OF FIGURES

Figure Page

Symmetric Encryption with Multiple Keys: Techniques and Applications

Introduction

Overview

This paper begins by presenting background information. The background section is intended for a technical reader who is not necessarily familiar with cryptography and cryptographic techniques.

Next, the paper describes the technique of extending block cipher modes of operation using multiple symmetric keys. In particular, it introduces the MK-ECB, MK-CTR, and MK-CBC modes of operation. It also describes the MK-CBC-MAC technique.

Possible applications of these techniques are presented, including establishing a covert channel, customizing message encryption at the block level, and expanding a string of pseudo-random numbers.

The paper concludes by describing simulation work, presenting comparisons to published works, discussing applicability of the techniques, and suggesting ideas for future work.

Notation and Conventions

Variables are emphasized in *italics* to distinguish them from text. Equation 1 indicates that C is the result of encrypting M with the key K. Equation 2 indicates that C is the result of encrypting M with the key selected by input b.

$$C = \text{Encrypt}\ [K, M] \quad (1)$$

$$C = \text{Encrypt}\ [Ks(b), M] \quad (2)$$

To concatenate two bit strings, we use the symbol ||. For example, 101 || 01011 = 10101011.

Background

Cryptography is the study of keeping messages secret. The original message is called plaintext. An encryption algorithm is used to transform the plaintext, using scrambling techniques, into ciphertext. To recover the original plaintext message, the ciphertext is decrypted. Decryption is the reverse process of encryption.

The scrambling techniques of modern encryption algorithms are published and analyzed; therefore, the security of these algorithms depends on a key. The key is chosen from a large set of values (called a keyspace), so that it is very difficult to guess the value of a particular key. For example, if a particular algorithm requires a 128-bit key, then the keyspace contains $2^{128} \approx 10^{38}$ possible keys.

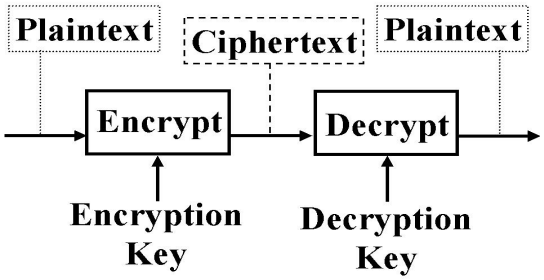

Figure 1: Encryption and Decryption

There are two main categories of encryption algorithms: asymmetric algorithms and symmetric algorithms.

Asymmetric Encryption Algorithms

Asymmetric encryption algorithms, such as RSA [RIVE78], use one key for encryption and a different key for decryption. The encryption key and the

decryption key are mathematically related, but it is computationally infeasible to determine the decryption key from the encryption key. The encryption key is typically called the public key, because it is openly shared with others. The decryption key is typically called the private key, because it must be kept a secret. Anyone can use your public key to encrypt a message that only you can decrypt.

Asymmetric encryption is also used to produce digital signatures. Algorithms that can be used to create digital signatures include RSA and DSA [NIST94]. To digitally sign a message, the signer encrypts with his private key. Only the signer can produce this particular signature, because only the signer knows his private key. The digital signature is verified by decrypting with the public key. Everyone who knows the signer's public key can verify the digital signature. Note that the signing process is the reverse of the encryption process.

Symmetric Encryption Algorithms

In traditional symmetric encryption algorithms, the same key is used for both encryption and decryption. The sender and the receiver must create and share this key before they can exchange encrypted messages. (Frequently, asymmetric encryption is used to share symmetric keys.) The key must be kept a secret because anyone who knows the key can decrypt the ciphertext and recover the plaintext. Symmetric encryption algorithms can be further classified into stream ciphers and block ciphers.

Stream Ciphers

Stream ciphers, such as RC4 [RIVE92], typically operate on a single bit or byte of plaintext at a time. A typical stream cipher combines a byte of plaintext with a byte from a pseudo-random stream of numbers, using the bitwise exclusive-OR (XOR) operation, described below. To recover the plaintext, the same pseudo-

random stream of numbers is needed. Stream ciphers must have a properly designed pseudo-random number generator (PRNG) to be secure. A stream cipher is usually faster than a block cipher, and is generally simpler to implement.

The XOR operation, symbolized by $\oplus$, is defined as the bitwise addition, modulo 2, of two bit strings of equal length. It has the following properties:

$$0 \oplus 0 = 0$$
$$0 \oplus 1 = 1$$
$$1 \oplus 0 = 1$$
$$1 \oplus 1 = 0$$

Note that $A \oplus A = 0$ and therefore $(A \oplus B) \oplus B = A$.

Pseudo-Random Number Generators

As mentioned above, pseudo-random number generators (PRNGs) are used in stream ciphers. They are also used to generate keys, prime factors, initialization vectors, and other random values. PRNGs are also known as deterministic random number generators (DRNGs).

There are many ways to construct a PRNG. For example, linear feedback shift registers (LFSRs) generate output sequences with good statistical properties, but an attacker can compute the entire output sequence if he knows enough output bits.

A block cipher such as AES or Triple-DES can also be used to construct a PRNG. In one configuration, the internal state S_i of the PRNG is encrypted with the key K to produce the next state S_{i+1}, which is also the output of the PRNG. That is,

$$S_{i+1} = \text{Encrypt}\,[K, S_i] = \text{Output}_{i+1} \qquad (3)$$

Another type of PRNG using DES is presented in the Digital Signature Standard [NIST2000]. The initialization values *T* and *C* are each 160 bits, and each *Ti* and *Ci* is 32 bits. The values *Yi,1* and *Yi,2* are each 32 bits. The DES encryption of *A* using key *B* is defined as *DES[B,A]*. The one-way function *G(T,C)* produces 160 bits of output.

$T = T_1 \| T_2 \| T_3 \| T_4 \| T_5$
$C = C_1 \| C_2 \| C_3 \| C_4 \| C_5$
$X_i = T_i \oplus C_i$
$B_1 = C_{((i+3) \bmod 5) + 1}$
$B_2 = C_{((i+2) \bmod 5) + 1}$
$A_1 = X_i$
$A_2 = X_{(i \bmod 5) + 1} \oplus X_{((i+3) \bmod 5) + 1}$
$Y_{i,1} \| Y_{i,2} = DES[(B_1 \| B_2), (A_1 \| A_2)]$
$Z_i = Y_{i,1} \oplus Y_{((i+1) \bmod 5)+1,2} \oplus Y_{((i+2) \bmod 5)+1,1}$
$G(T,C) = Z_1 \| Z_2 \| Z_3 \| Z_4 \| Z_5$

The intermediate values of the algorithm are shown in the following table.

i	B1	B2	A1	A2	Y = Y1,1 ‖ Y1,2	Zi
1	C5	C4	T1 ⊕ C1	(T2⊕ C2)⊕ (T5⊕ C5)	DES [(C5‖C4), (T1 ⊕ C1)‖ (T2⊕ C2)⊕ (T5⊕ C5)]	Y1,1⊕ Y3,2 ⊕ Y4,1
2	C1	C5	T2 ⊕ C2	(T3⊕ C3)⊕ (T1⊕ C1)	DES [(C1‖C5), (T2 ⊕ C2)‖ (T3⊕ C3)⊕ (T1⊕ C1)]	Y2,1⊕ Y4,2 ⊕ Y5,1
3	C2	C1	T3 ⊕ C3	(T4⊕ C4)⊕ (T2⊕ C2)	DES [(C2‖C1), (T3 ⊕ C3)‖ (T4⊕ C4)⊕ (T2⊕ C2)]	Y3,1⊕ Y5,2 ⊕ Y1,1
4	C3	C2	T4 ⊕ C4	(T5⊕ C5)⊕ (T3⊕ C3)	DES [(C3‖C2), (T4 ⊕ C4)‖ (T5⊕ C5)⊕ (T3⊕ C3)]	Y4,1⊕ Y1,2 ⊕ Y2,1
5	C4	C3	T5 ⊕ C5	(T1⊕ C1)⊕ (T4⊕ C4)	DES [(C4‖C3), (T5 ⊕ C5)‖ (T1⊕ C1)⊕ (T4⊕ C4)]	Y5,1⊕ Y2,2 ⊕ Y3,1

This technique uses portions of *C* as key material to encrypt specific combinations of portions of *C* and *T*. Note that the key material is combined in different ways for each iteration of the function.

Block Ciphers

Block ciphers, such as DES [NIST93] and AES [NIST2001a] operate on groups of plaintext bits called blocks. DES, first published by NIST as a standard in 1977, uses a 56-bit key to operate on 64-bit plaintext blocks, producing 64-bit ciphertext blocks. AES, published by NIST about twenty years later, operates on 128-bit plaintext blocks to produce 128-bit ciphertext blocks. AES keys can be 128 bits,

192 bits, or 256 bits in length. The common implementation of AES uses a 128-bit key.

Modern block ciphers use a combination of substitutions, permutations, shifts, swaps, XOR operations, and other linear and nonlinear operations to transform each block of plaintext. Block ciphers have a wide range of applications; they are generally considered the workhorses of cryptography.

Modes of Operation

A 2001 NIST publication [NIST2001b] recommends five modes of operation to be used with a symmetric key block cipher algorithm such as AES: ECB, CTR, CBC, CFB, and OFB. These modes are summarized in the following sections.

The plaintext consists of k blocks. The i^{th} block of the plaintext is denoted Mi. Therefore, when the original plaintext message M is divided into blocks it can be represented as

$$M = M_1, M_2, M_3, \ldots M_k \quad (4)$$

Similarly, the ciphertext will consist of k blocks. The ciphertext C can be represented as

$$C = C_1, C_2, C_3, \ldots C_k \quad (5)$$

In addition, the CBC, CFB, and OFB modes include an input value called the initialization vector, or *IV*. The *IV* does not need to be kept a secret. For the CBC and CFB modes, the *IV* should be unpredictable for each execution of the encryption process. For the OFB mode, each execution of the encryption process needs a unique *IV*.

The variable *SR*, used in the CFB and OFB modes, represents the contents of a 64-bit shift register. The contents of *SR* are shifted *s* bits to the left in each iteration, and a *s*-bit value is shifted into the *s* vacant least significant bit positions.

We also define the function $LSB_s( )$, which returns the *s* least significant bits of the argument, and $MSB_s( )$, which returns the *s* most significant bits of the argument. For example, $LSB_2(01110010) = 10$, and $MSB_3(01110010) = 011$.

ECB Mode

The Electronic Codebook (ECB) mode of operation is simple. It is described by Equation 6 and Figure 2. Each block of plaintext is encrypted independently. However, for a particular key, there is a direct mapping between each unique block of plaintext and the corresponding block of ciphertext. We can imagine a huge codebook, listing the ciphertext for each block of plaintext.

$$Ci = \text{Encrypt}[key, Mi] \qquad (6)$$
$$Mi = \text{Decrypt}[key, Ci]$$

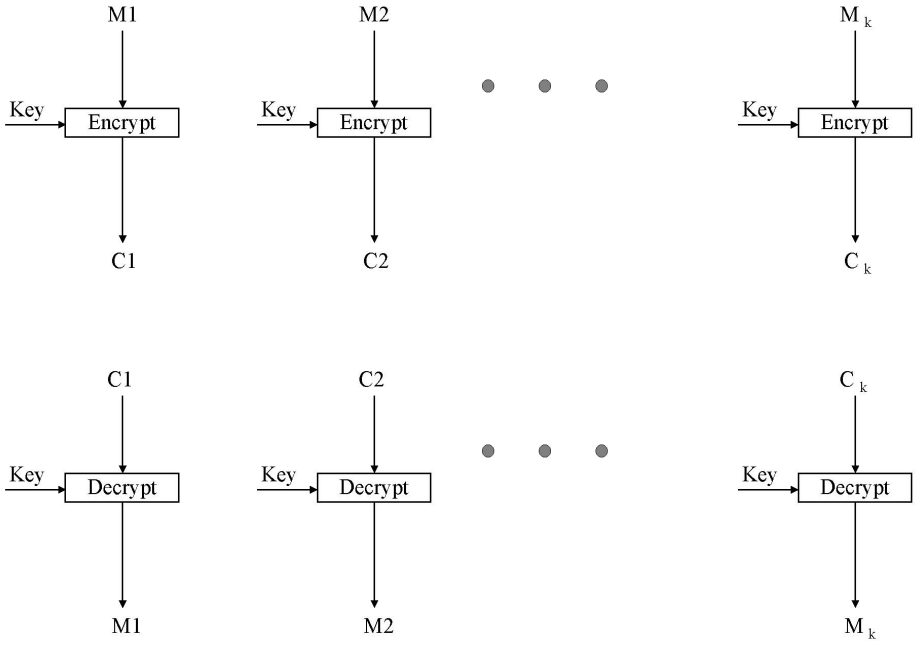

Figure 2: ECB Mode

For a short message, ECB is a good mode to use. The sender can encrypt multiple plaintext blocks in parallel, and the receiver can decrypt multiple ciphertext blocks in parallel. Blocks may be encrypted or decrypted out of order. An error in one block does not propagate. However, because there is a direct correlation between the plaintext and the ciphertext, it is possible for an attacker to exploit patterns and regularities in the plaintext. Plaintext blocks could also be removed, repeated, or exchanged. This implies that insertion, deletion, and replay attacks (described later in this section) are possible.

CTR Mode

Counter mode of operation (CTR) uses a counter output the same size as a block. For each encrypted block, the counter value must be different. (Typically, the counter is incremented for each iteration, as shown below.) The same sequence of counter values is used for encryption and decryption. CTR mode is described by Equation 7 and Figure 3. The sender encrypts the counter value with the key and

the result is XORed with the plaintext block to produce the ciphertext block. The receiver encrypts the same counter value with the key and XORs the result with the ciphertext block to produce the plaintext block. Recall that, due to the properties of the XOR operation, the recovered plaintext block will be identical to the original plaintext block if there are no errors in transmission.

$$Ci = \text{Encrypt}[key, counter+(i-1)] \oplus Mi \qquad (7)$$
$$Mi = \text{Encrypt}[key, counter+(i-1)] \oplus Ci$$

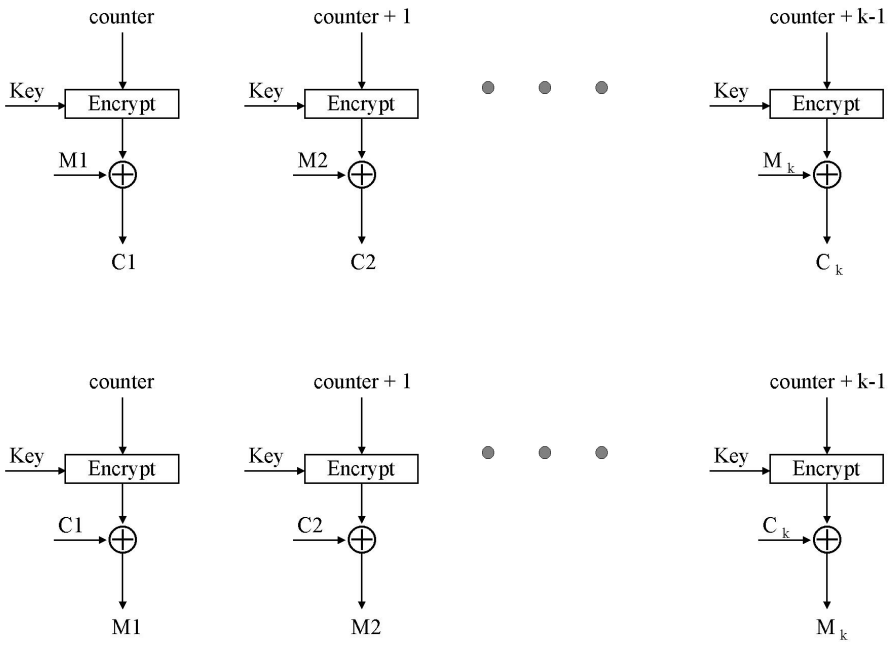

Figure 3: CTR Mode

Even if the same plaintext block is repeated, the ciphertext blocks will be different. As in ECB mode, the sender can encrypt multiple plaintext blocks in parallel, and the receiver can decrypt multiple ciphertext blocks in parallel. Blocks may be encrypted or decrypted out of order. An error in one block does not propagate. To further improve efficiency, the intermediate values resulting from the encryption step may be pre-computed. In addition, note that only the encryption algorithm is needed.

CBC Mode

Cipher Block Chaining (CBC) is a common mode of operation. CBC mode is described by Equation 8 and Figure 4. The sender XORs the current plaintext block with the previous ciphertext block (or the *IV*) and the resulting block is encrypted with the key to produce the current ciphertext block. The receiver decrypts the current ciphertext block with the key, and the resulting block is XORed with the previous ciphertext block (or the *IV*) to recover the current plaintext block. Both the sender and the receiver must know the *IV*.

$$
\begin{aligned}
&C1 = \text{Encrypt}\,[key, (M1 \oplus IV)] && (8)\\
&Ci = \text{Encrypt}\,[key, (Mi \oplus C_{i-1})] \quad (\text{for } k \geq i \geq 2)\\
&M1 = \text{Decrypt}\,[key, C1] \oplus IV\\
&Mi = \text{Decrypt}\,[key, Ci] \oplus C_{i-1} \quad (\text{for } k \geq i \geq 2)
\end{aligned}
$$

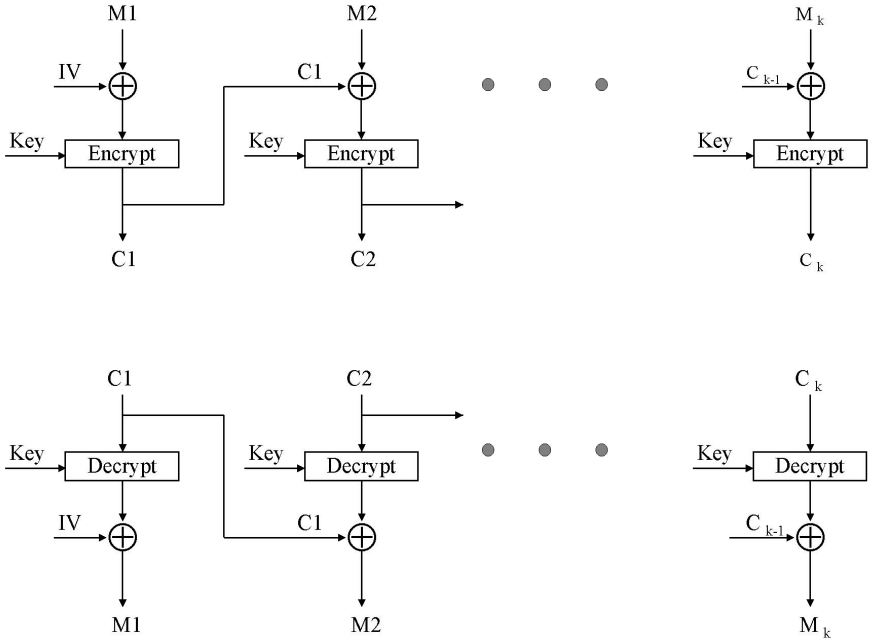

Figure 4: CBC Mode

As in CTR mode, even if the same plaintext block is repeated, the ciphertext blocks will be different. The ciphertext of any given plaintext block is a function of all the previous plaintext blocks in the message. The sender can not encrypt multiple plaintext blocks in parallel, but the receiver can decrypt multiple ciphertext blocks in parallel. Note that a transmission error in one ciphertext block affects the recovery of the current plaintext block and the next plaintext block.

CFB Mode

Cipher Feedback mode (CFB) is described by Equation 9 and Figure 5. The parameter L is the block size of the cipher, and the parameter s is chosen such that $L \geq s \geq 1$. Typical values of s are 1, 8, 64, and 128. Each plaintext segment is s bits and each ciphertext segment is s bits. The sender encrypts the contents of the shift register with the key, and selects the s most significant bits of the result to XOR with the current plaintext segment to produce the current ciphertext segment. The receiver encrypts the contents of the shift register with the key, and selects the s most significant bits of the result to XOR with the current ciphertext segment to recover the current plaintext segment. Both the sender and the receiver must know the *IV*.

$$
\begin{aligned}
&SR_1 = IV \qquad (9)\\
&SR_i = LSB_{L-s}(SR_{i-1}) \,||\, C_{i-1} \quad (\text{for } k \geq i \geq 2)\\
&C_i = MSB_s(\text{Encrypt } [\text{key}, SR_i]) \oplus M_i\\
&M_i = MSB_s(\text{Encrypt } [\text{key}, SR_i]) \oplus C_i
\end{aligned}
$$

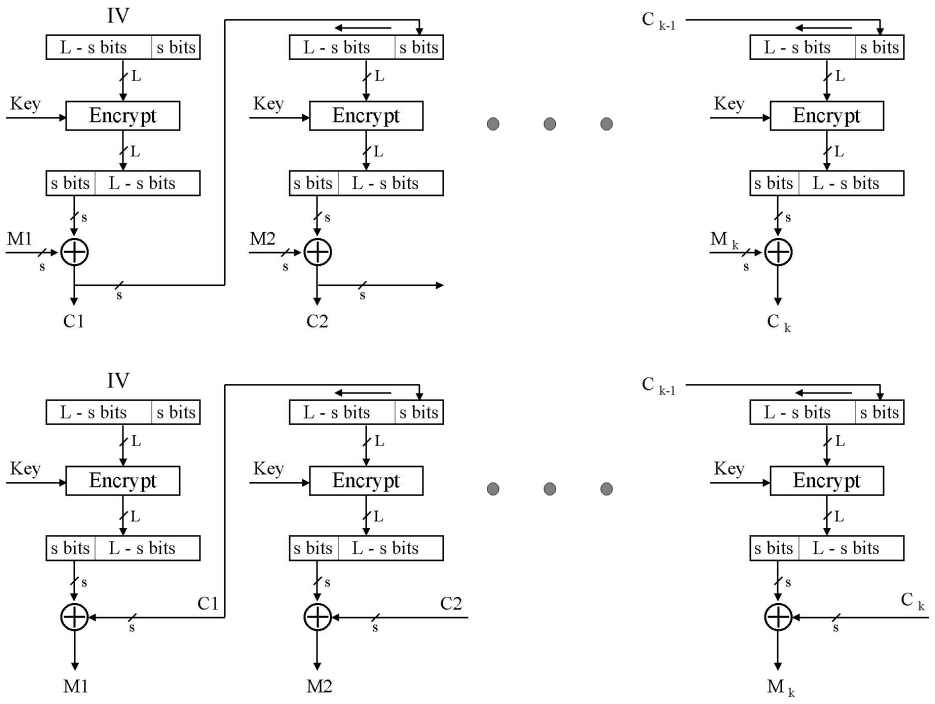

Figure 5: CFB Mode

As in CTR and CBC mode, even if the same plaintext segment is repeated, the ciphertext segments will be different. Like CBC mode, the ciphertext of any given plaintext segment is a function of all the previous plaintext segments in the message. The sender can not encrypt multiple plaintext segments in parallel, but the receiver can decrypt multiple ciphertext segments in parallel. Note that a transmission error in one ciphertext segment affects the recovery of the next *L/s* (rounded up) ciphertext segments.

OFB Mode

Output Feedback mode (OFB) is described by Equation 10 and Figure 6. As in CFB mode, the parameter L is the block size of the cipher, and the parameter s is chosen such that $L \geq s \geq 1$. Typical values of s are 1, 8, 64, and 128. The form of OFB is similar to CFB, except that the shift register is updated with the most significant bits of the output of the previous encryption, instead of with the previous ciphertext segment.

$$SR_1 = IV \qquad (10)$$
$$SR_i = LSB_{L-s}(SR_{i-1}) \,||\, MSB_s(\text{Encrypt } [key, SR_{i-1}]) \qquad (\text{for } k \geq i \geq 2)$$
$$C_i = MSB_s(\text{Encrypt } [key, SR_i]) \oplus M_i$$
$$M_i = MSB_s(\text{Encrypt } [key, SR_i]) \oplus C_i$$

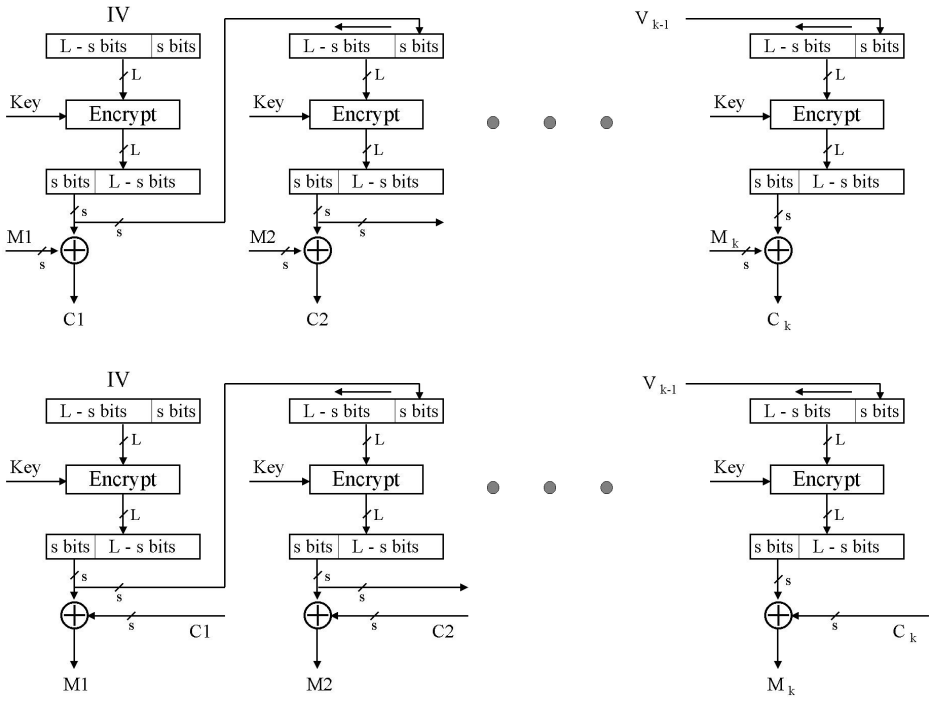

Figure 6: OFB Mode

As in the CTR, CBC, and CFB modes, even if the same plaintext segment is repeated, the ciphertext segments will be different. Unlike CBC and CFB, the ciphertext of any given plaintext segment is not a function of all the previous plaintext segments in the message. If the *IV* is known, the output of the encryption step can be computed in advance. Therefore, the sender can encrypt multiple plaintext segments in parallel, and the receiver can decrypt multiple ciphertext segments in parallel. Note that a transmission error in one ciphertext segment only affects the recovery of the current plaintext segment.

The following table summarizes certain properties of the ECB, CTR, CBC, CFB, and OFB modes of operation.

Mode	Hides Plaintext Statistics?	C_i depends on M_1 to M_{i-1} blocks?	Parallel Encrypt?	Parallel Decrypt?	Ci Transmission Error Affects...?
ECB	No	No	Yes	Yes	Current Mi only
CTR	Yes	No	Yes	Yes	Current Mi only
CBC	Yes	Yes	No	Yes	M_i and M_{i+1}
CFB	Yes	Yes	No	Yes	M_i to $M_{i+(L/s)}$
OFB	Yes	No	Yes	Yes	Current Mi only

Note that bit errors in the *IV* also affect the receiver's decryption process [NIST2001b]. Also note that the speed of each of these modes is essentially the speed of the block cipher, because the XOR operation takes negligible time in comparison with an encryption or decryption operation.

Other Modes

Other modes of operation include Plaintext Block Chaining (PBC) and Plaintext Feedback (PFB). The PBC mode is similar to CBC, except that the plaintext blocks (instead of the ciphertext blocks) are passed to the next iteration. That is, the previous plaintext block is XORed with the current plaintext block. The PFB mode is similar to CFB, except that the plaintext blocks (instead of the ciphertext blocks) are fed back to the previous iteration. The PBC and PFB modes are not common.

An interesting but uncommon mode [JANS88] is called Output Feedback with a Nonlinear Function (OFBNLF). The OFBNLF mode is based on ECB and OFB. Note that the key changes with every block. The OFBNLF mode is described by Equation 11 and Figure 7.

$$K_i = \text{Encrypt}\ [K, K_{i-1}] \qquad (11)$$
$$C_i = \text{Encrypt}\ [K_i, M_i]$$
$$M_i = \text{Decrypt}\ [K_i, C_i]$$

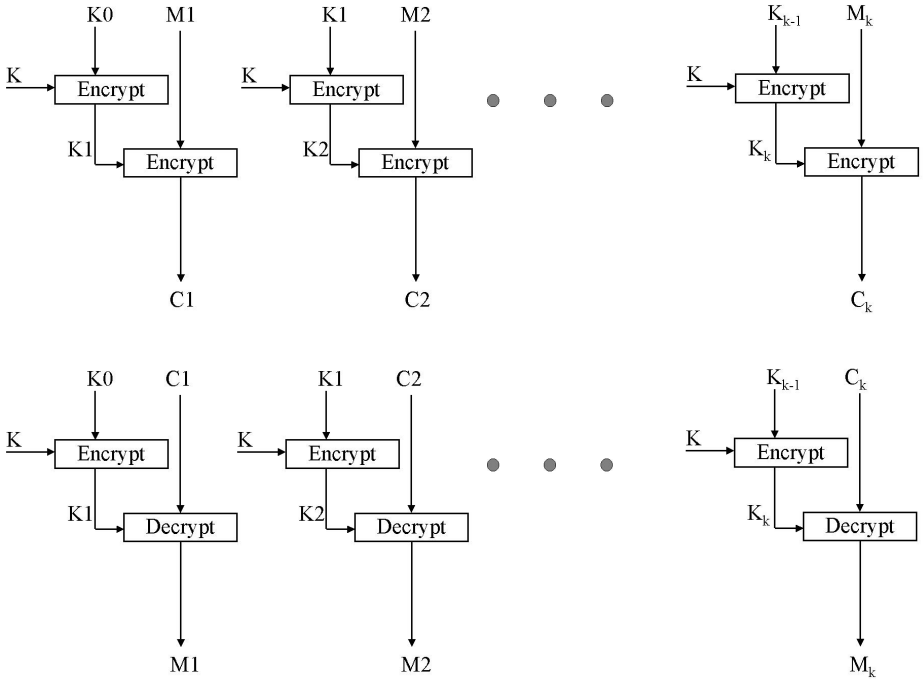

Figure 7: OFBNLF Mode

A transmission error in one ciphertext segment only affects the recovery of the current plaintext segment. However, this mode can not recover from even a single addition or deletion. In addition, this mode would be slow because the key must be rescheduled for the block cipher for each block.

Message Authentication

Message authentication is used to verify that a message received from an alleged source has not been altered. A (non-cryptographic) checksum or an error detecting code is designed to detect accidental transmission errors in the data. Other

techniques can detect not only transmission errors, but also intentional modifications of the data by an active attacker.

There are many techniques used to authenticate a message. For example, in symmetric encryption, the ciphertext of an entire message serves to authenticate the message, because only someone who knows the shared key can compute the ciphertext from the plaintext. However, the receiver must be able to determine that the plaintext is valid, and this would not be obvious if the plaintext was a string of random numbers. Also, because the receiver shares the key with the sender, the receiver could forge the message. Further, the sender could deny sending the message, and claim that the receiver sent it.

A digital signature (described previously) also provides message authentication, but has several advantages over the technique of only using symmetric encryption. Only the signer knows his private key, and therefore only the signer could have produced the signature for a given message. Anyone who has the signer's public key can verify the signature. Therefore, the sender can not deny signing the message, and the receiver can not forge the message.

Another technique to authenticate a message is called a message authentication code (MAC), also known as a cryptographic checksum. The process for computing the MAC is a one-way function. A shared secret key is used to generate a fixed-size output value called the MAC from an input message of any length. Only someone who knows the key can verify the MAC.

A MAC can be used be used to indicate whether stored files have been altered. It can also be used to authenticate messages transmitted from sender to receiver.

In the latter case, suppose the sender transmits a message and its MAC to the receiver. The receiver performs the same calculation on the message using the shared key, and compares the result to the MAC it received. If there is a match,

the receiver is assured that the message is from the sender, and that the message was not altered in transit.

It is possible for an attacker to compute the MAC of a known message for each key in a keyspace. Let the MAC for M_1 be called MAC_1. The attacker computes $MAC_i = function[K_i, M_1]$ for all possible K_i, where *function* is the process for computing the MAC for the message using the key. Because many messages produce the same MAC, the attacker can find $MAC_i = MAC_1$. This type of attack, called a collision attack, may be less work than a brute force attack on the key. Therefore, the MAC function should be chosen so that it is difficult to find a collision.

CBC-MAC

CBC-MAC [ISO89] is an algorithm used to generate a MAC. The CBC mode of operation is applied to the plaintext message using an initialization vector (*IV*) of zero. The final block of the resulting CBC output (*Ck*) serves as the MAC. This process is described by Equation 12 and Figure 8.

$$C1 = \text{Encrypt}\,[\text{key}, (M1 \oplus 0)] \qquad (12)$$
$$Ci = \text{Encrypt}\,[\text{key}, (Mi \oplus C_{i-1})] \quad (\text{for } k \geq i \geq 2)$$
$$MAC = Ck$$

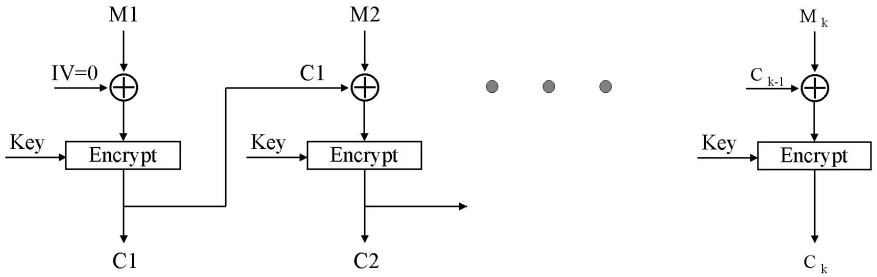

Figure 8: CBC-MAC

A drawback of this method is that the sender and the receiver share the key used to authenticate the message. Suppose the sender transmits a message and the MAC for that message. The receiver can generate a different message with the same MAC. All the receiver needs to do is decrypt in the reverse direction [SCHN96].

CCM

In some applications, it may be desirable to authenticate the message, but not encrypt the message. In other applications, both authentication and encryption may be needed. The Counter with Cipher Block Chaining-Message Authentication Code (CCM) technique is used to both authenticate and encrypt a message.

In CCM [NIST2004], the same key *K* is used for both the CTR and CBC-MAC mechanisms. The message, associated data (such as headers), and a nonce are concatenated to form *P*. Note that the nonce should be unique but it does not need to be random; this means that any two different applications of CCM using the same key should have different nonce values. CBC mode is applied to *P* to produce the MAC. The MAC is concatenated with *P* to form *M*. Therefore, the size of the transmitted message is increased by the size of the MAC. Next, *M* is encrypted using CTR mode to generate the ciphertext *C*. The ciphertext is decrypted by the receiver to recover the plaintext, including the MAC. The MAC is verified by applying CBC mode to the message, the associated data, and the nonce. If verification is successful, the receiver believes the message originated from someone who knows the key. This process is described below.

Sender:
P = message || associated data || nonce
MAC = CBC-MAC [K, P]
M = P || MAC
C = CTR-Encrypt [K, M]

Receiver:
M = CTR-Decrypt [K, C]
M = P || MAC
MAC' = CBC-MAC [K, P]
Test if MAC' = MAC

Combining Block Ciphers

There are different ways to combine block ciphers to produce new block ciphers. In this section, we consider several different techniques. For instance, interleaving combines block ciphers using the same key (and typically the same mode of operation) but different *IV*s. Multiple encryption uses multiple keys to encrypt each block multiple times. Cascading is similar to multiple encryption, but uses multiple ciphers to encrypt.

Interleaving

Many common modes of operation (particularly CBC, CFB, and OFB) rely on the previous encryption result to generate the current encryption result. This means that even if multiple instances of encryption hardware are available, only one is used at a time.

Interleaving is a technique to encrypt blocks in parallel, which can lead to greater efficiency. Interleaving divides the blocks of a message into sets. All the blocks of a message are encrypted with the same key; however, the blocks in one set must have a different *IV* than the blocks in another set. This means that the length of the

combined *IV* for all the sets has increased proportionally to the number of sets. The output streams are interleaved before they are transmitted.

Consider, as an example, that a particular computer system has two different chips to perform block cipher encryption. Both chips use the same key. Chip 1 encrypts all the odd numbered blocks of the message with *IV1*, and chip 2 encrypts all the even numbered blocks of the message with *IV2*. The combined *IV* is twice the length of a single *IV*. The output is one block from chip 1, one block from chip 2, one block from chip 1, and so on.

Double and Triple Encryption

Double encryption is vulnerable to a meet-in-the-middle attack (described later), and offers little benefit over single encryption. Double encryption is typically not used.

Triple encryption, however, is common. One standardized implementation of triple encryption, called encrypt-decrypt-encrypt (EDE) mode, was designed to be compatible with the conventional single encryption implementation of DES. As described in Equation 13, the plaintext is encrypted with *K1*, then the result is decrypted with *K2*, and finally the result is encrypted with *K1* again.

$$\begin{aligned} C &= \text{Encrypt}[K1, \text{Decrypt}[K2, \text{Encrypt}[K1, M]]] \\ M &= \text{Decrypt}[K1, \text{Encrypt}[K2, \text{Decrypt}[K1, C]]] \end{aligned} \qquad (13)$$

EDE triple encryption is also vulnerable to a type of meet-in-the-middle attack [MERK81]. However, this attack requires 2^n time, 2^n memory blocks, and 2^m chosen plaintexts. Here, n is the length of the key in bits and m is the size of a block in bits. There is also a known-plaintext attack [OORS91] which only requires $(2^{n+m})/p$ time, p memory storage space, and p known plaintexts. With enough known plaintexts, this attack can be faster than an exhaustive search. (Known-plaintext and chosen-plaintext attacks are described in a later section.)

Triple encryption with three independent keys is generally considered more secure than EDE triple encryption. Using three independent keys, triple encryption can be attacked in 2^{2n} time, using 2^n memory blocks [MERK81].

Other Multiple Encryption Techniques

Double OFB/CTR [SCHN96] is described by Equation 14. A block cipher is used to generate two keystreams, and these keystreams are used to encrypt the plaintext. Two counters, *counter1* and *counter2*, are needed. The keys *K1* and *K2* are independent.

$$S_i = \text{Encrypt}\ [K1, (S_{i-1} \oplus \text{counter1+i-1})] \qquad (14)$$
$$T_i = \text{Encrypt}\ [K2, (T_{i-1} \oplus \text{counter2+i-1})]$$
$$C_i = M_i \oplus S_i \oplus T_i$$

Another method [BLAZ93, BLAZ94] is used in the UNIX Cryptographic File System (CFS) to encrypt multiple messages of a fixed length. Two keys are used, *K1* and *K2*. A mask is created using *K1* and the encryption algorithm. The plaintext is XORed with the mask. The result is encrypted in ECB mode using *K2* and the encryption algorithm. This method is based on the OFB and ECB modes of operation.

To make it difficult for an attacker to obtain a plaintext-ciphertext pair, a technique called whitening is used. Some key material (called *K1*) is XORed with the input to the block cipher, and the output is XORed with some other key material (called *K3*). This technique is described by Equation 15.

$$C = K3 \oplus \text{Encrypt}\ [K2, (M \oplus K1)] \qquad (15)$$
$$M = K1 \oplus \text{Decrypt}\ [K2, (C \oplus K3)]$$

This whitening technique is not vulnerable to a meet-in-the-middle attack. An attack requires 2^{n+m+1} operations with three known plaintexts. Recall that n is the

length of the key in bits and *m* is the size of a block in bits. This is an easy way to increase the computational security of a block algorithm.

Cascading

A cascade of different ciphers is at least as hard to break as any of its component ciphers, using a chosen-plaintext attack [KNUD94]. It is possible that subtle interactions between different ciphers could diminish security; however, this is unlikely if independent keys are used for each cipher in the cascade. An example of cascading is described by Equation 16.

$$\begin{aligned} V1 &= \text{Encrypt}\,[K1, W] \quad (\text{cipher1}) \\ V2 &= \text{Encrypt}\,[K2, (M \oplus W)] \quad (\text{cipher 2}) \\ C &= V1 \,||\, V2 \end{aligned} \tag{16}$$

In this example [SCHN96], *W* is a random string of bits the same length as the plaintext message *M*. The keys *K1* and *K2* are independent. The intermediate result *V1* is the result of encrypting with *cipher1*. The intermediate result *V2* is the result of encrypting with *cipher2*. The ciphertext *C* is *V1* and *V2*, which could be twice the size of the plaintext message.

Multiple-Key Protocols

Suppose that a message encrypted with *K1* is decrypted with *K2* and *K3*. Similarly, a message encrypted with *K2* and *K3* is decrypted with *K1*. This variant of asymmetric encryption is described in [BOYD88]. The table shows possible encryption and decryption combinations for three keys.

Encrypt with Key(s):	Decrypt with Key(s):
K1	K2 and K3
K2	K1 and K3
K3	K1 and K2
K1 and K2	K3
K1 and K3	K2
K2 and K3	K1

Suppose that keys are distributed to the members of a group as shown below.

Key Holder	Has the Key(s):	Decrypts from:	Encrypts for:
A	K1	F	F
B	K2	E	E
C	K3	D	D
D	K1 and K2	C, E, F	C, E, F
E	K1 and K3	B, D, F	B, D, F
F	K2 and K3	A, D, E	A, D, E

Observe that this key distribution forms subgroups of the group members. Group member F can encrypt a message with *K2* and *K3*, with *K3* only, or with *K2* only. This means that F can send an encrypted message to A, D, or E, but not to B or C. However, B and C can work together to impersonate F.

The benefit of this technique becomes apparent when there are many keys, instead of only *K1*, *K2* and *K3*. Without this technique, for *n* members in a group, on the order of 2^n different keys are needed to communicate with each possible subgroup. In contrast, this technique only needs *n* different keys to communicate with each possible subgroup. However, unless the message indicates in unencrypted form which subgroup is the intended recipient, each member of the group would have to try all possible combinations of their keys. Also, every member of the group stores a large amount of key material.

Cryptanalysis

In this section, we consider several types of attacks. We focus on attacks against the confidentiality of a message and attacks against the integrity of a message.

Attacks against the Confidentiality of Messages

An attack against the confidentiality of a message is an attack against the cipher itself. The most basic goal of a cipher is to keep plaintext messages secret. The objective of the attacker is to recover the plaintext.

It is reasonable to assume that an attacker has complete access to the communication channel between the sender and the receiver. That is, the attacker sees the ciphertext and knows the details of the algorithm(s) used to encrypt and decrypt.

The ultimate prize for an attacker is to find the secret key. With this key, the attacker has all the information necessary to recover the plaintext. This is known as a total break. Four general types of attacks could potentially lead to a total break. These are called ciphertext-only attacks, known-plaintext attacks, chosen-plaintext attacks, and chosen-ciphertext attacks.

In a ciphertext-only attack, the attacker only has the ciphertext of several messages to work with. Recall our assumption that the attacker knows the ciphertext. This type of attack is the most difficult, because the attacker has the least amount of information.

In a known-plaintext attack, the attacker knows the plaintext that corresponds with the ciphertext. Known-plaintext attacks are common. For example, many plaintext messages have standard beginnings and endings that can be guessed by the attacker. If an attacker has some known plaintexts, he could do an exhaustive

search over the entire keyspace (also known as a brute force attack) to find the secret key.

In a chosen-plaintext attack, the attacker can choose the plaintext to encrypt. A chosen-plaintext attack can be more powerful than a known-plaintext attack because the attacker can choose specific plaintext blocks that may help him discover the key.

In a chosen-ciphertext attack, the attacker can choose the ciphertext blocks to decrypt, and can see the resulting plaintext. For example, consider that an attacker has access to a tamper-proof decryption machine. The attacker can recover the plaintext using the machine, but can not see inside the machine to deduce the key.

Attacks against the Integrity of Messages

In addition to providing confidentiality, cryptography (and coding theory) can be used to provide integrity. The sender wants his original message to get to the receiver intact. Common mechanisms to check the integrity of a message include digital signatures, message authentication codes, and error detection/correction techniques.

A communication channel may be noisy; it may introduce errors into the ciphertext blocks. Integrity checking mechanisms may be able to detect and correct these errors.

It is also possible that an active attacker could purposefully and maliciously modify the ciphertext during transmission from sender to receiver. This active attacker is sometimes called a man-in-the-middle. He positions himself between the sender and the receiver and intercepts the ciphertext. This attacker could change a ciphertext block, insert or delete ciphertext blocks, rearrange the order of the ciphertext blocks, and substitute illegitimate blocks for legitimate blocks. The

attacker could also perform a replay attack (described below). A sophisticated attacker might even be able to ensure that the integrity check still works after his modifications.

Replay Attacks

In a replay attack, the attacker makes an exact copy of a message (including any integrity checking mechanism) and later sends the copy. For example, the attacker records the ciphertext corresponding to the message "Transfer $1,000 to my bank account." The attacker then replays this message, and gets rich. Timestamps are frequently used to frustrate replay attacks. However, an attacker could still replay a timestamped message within the valid time interval.

Meet-in-the-Middle Attacks

The meet-in-the-middle attack [MERK81] was developed to break a block cipher double encryption scheme in 2^{n+1} encryptions instead of 2^{2n} encryptions, where n is the length of the encryption key in bits. It assumes that the attacker knows the plaintext-ciphertext pairs (*M1*, *C1*) and (*M2*, *C2*) such that

$$\begin{aligned} C1 &= \text{Encrypt}[K2, \text{Encrypt}[K1, M1]] \\ C2 &= \text{Encrypt}[K2, \text{Encrypt}[K1, M2]] \end{aligned} \qquad (17)$$

For each possible key *K*, the attacker computes *Encrypt [K, M1]* and stores the result. Then the attacker computes *Decrypt [K, C1]* for each possible key *K*. If *Encrypt [K, M1]* matches *Decrypt [K, C1]*, then it is possible that the attacker has found *K1* and *K2*. To check, the attacker encrypts *M2* with *K1* and *K2*. If the result is *C2* then the attack was probably a success.

A drawback of the meet-in-the-middle attack is that it requires 2^n blocks of memory. For a 56-bit algorithm such as DES, this means 2^{56} 64-bit blocks are

needed, which is on the order of 10^{16} bytes. For 10-round AES, this means 2^{128} 128-bit blocks are needed, which is on the order of 10^{38} bytes.

Other Types of Attacks

Many different types of attacks are possible. These range from passive eavesdropping to malicious intervention. It is important to remember that even participants in a protocol (such as the sender or the receiver) could cheat.

Further, it is possible to observe patterns in information, even if the message itself is encrypted. For example, the IP addresses used to send information from sender to receiver on the Internet are not encrypted. (This is analogous to putting an address on an envelope before you mail a letter. The letter is not delivered if the postal service can not read the address.) By simply observing the source and destination IP addresses, timing, and frequency of messages sent across the public network, it is possible to gain a significant amount of information.

In addition, it is often the case that human users are the weak link in a system. Users share their passwords, or write them down where an attacker can find them. Users could be coerced to give away secret information, including keys. They may even give out information voluntarily, if they do not understand that it should be kept secret, or if they intend to cheat. It may not be possible for technology to compensate for the actions of users.

Particularly because there are so many possible attacks, it is important to have detailed analysis and peer review of any new ideas in the field of information security.

Subliminal Channels

Steganography vs. Cryptography

There are two basic ways to hide a message, steganography and cryptography. As we have discussed, cryptography does not conceal the existence of a message, but instead transforms the plaintext message using scrambling techniques into random-looking ciphertext. The intent is to prevent an attacker or eavesdropper from seeing the plaintext message while it is transmitted over an insecure channel. Recall that cryptographic algorithms (such as DES, AES, RC4, and RSA) are published. The security of the message does not depend on the secrecy of the algorithm; the security depends on a key.

In contrast, steganography conceals the existence of a message. Typically, a secret message is hidden within another message. The steganographic algorithm, or method for hiding the secret message, is often not disclosed to outsiders. This concept has been utilized throughout history [KAHN67]. For example, the second letter of each word in a sentence could spell out a secret message. A secret message could be hidden in a crossword puzzle, such that when the puzzle is solved the message is revealed. A particularly inventive technique involved shaving a messenger's head, tattooing the secret message on the messenger's scalp, and waiting for the hair to grow back before sending the messenger on his way. A relatively recent technique replaces the least significant bits of a digital image file with the bits of the secret message. To a casual observer the digital image does not appear to be modified, but the embedded message can be extracted if the steganographic algorithm is known.

The intersection between steganography and cryptography is particularly interesting. Secret messages can be encrypted using cryptographic algorithms

before being embedded in another message using steganography. This way, even if the steganographic algorithm is revealed and the existence of the hidden message is discovered, the hidden message can not be deciphered without the cryptographic key.

Further, cryptographic algorithms and protocols may themselves contain opportunities for hiding a secret message within a normal-looking message. This technique is called a covert channel, or a subliminal channel. A specific concern is that subliminal channels in cryptographic algorithms or protocols might be used to leak secret information, such as key material, without being detected.

Many techniques used to create subliminal channels extend beyond simple steganography. Subliminal channels typically require a shared key between the sender and the receiver. The security of the subliminal channel depends on the secrecy of this key. Techniques for constructing subliminal channels may be published, without diminishing the effectiveness of the techniques.

The Prisoner's Problem

In 1983, Gustavus Simmons introduced the concept of a subliminal channel [SIMM84]. The concept was based on the following abstract model.

Two accomplices have been arrested for a crime, and will be imprisoned in different cells. The warden permits them to exchange messages only if the information contained in the messages is completely open to him. The warden wants to be aware of any plans for escape, and wants the opportunity to substitute fabricated or modified messages for genuine messages. In order to plan their escape, the prisoners must find a way to communicate secretly by establishing a subliminal channel. Also, because the prisoners anticipate deception by the warden, they only want to exchange messages that the other may authenticate.

The prisoners solve this problem by subverting the authentication mechanism. If the prisoners have *m* bits of information to exchange and *r* bits of authentication, then a total of *(m+r)* bits must be communicated. Without the knowledge of the warden, the prisoners give up some of their ability to authenticate in exchange for the ability to communicate secretly. To communicate *s* bits of secret information, the prisoners now have *(r-s)* bits available for authentication. This means it is more likely that the warden can succeed in deceiving the prisoners.

Simmons' paper describes two examples [SIMM84] illustrating this concept. One example also demonstrates that detecting the existence of the subliminal channel could be as computationally difficult as breaking the authentication algorithm. While Simmons' examples were intended only as a proof of concept, it was soon demonstrated that digital signature algorithms could be used to create subliminal channels.

Subliminal Channels in Digital Signatures

Both the Ong-Schnorr-Shamir identification scheme [SIMM84] and the El Gamal digital signature algorithm [SIMM94a] can be used to create a subliminal channel [SCHN96]. A problem with these schemes is a protocol weakness [SIMM86] that allows the subliminal receiver to impersonate the subliminal transmitter. The subliminal transmitter must trust the subliminal receiver with his private key. Consequently, the subliminal receiver could forge the digital signature of the subliminal transmitter.

A subliminal channel could also be added to the ESIGN digital signature scheme [SIMM86]. This subliminal channel has the advantage that the subliminal receiver can not impersonate the subliminal transmitter. Only a part of the subliminal transmitter's private key is shared with the receiver. It is computationally infeasible for the subliminal receiver to recover the subliminal transmitter's private key.

The Digital Signature Algorithm (DSA) actually has several possibilities for subliminal channels [SIMM94a, SIMM86]. Each of these possibilities relies on the subliminal transmitter's ability to choose the DSA parameter called *k*. The parameter *k* is specified as a 160-bit random number [NIST94].

The simplest subliminal channel in DSA allows the subliminal transmitter to communicate a 160-bit message to the subliminal receiver by choosing a particular value of *k*. Because the *k* parameter should appear to be random, the subliminal message should be encrypted using a non-repeating set of truly random values called a one-time pad. The one-time pad must be shared between the subliminal transmitter and the subliminal receiver, and can only be used once. Another drawback of this technique is that the subliminal transmitter's private key is shared with the subliminal receiver.

DSA also has a subliminal channel that does not require the subliminal transmitter's private key to be shared with the subliminal receiver. To send a single bit of subliminal information per signed message, the subliminal transmitter and the subliminal receiver agree on a shared secret key for the subliminal key. This shared secret key is a prime that we will call *P* (note that this *P* is not defined by DSA). To communicate the subliminal message "0", the subliminal transmitter chooses a particular value for the *k* parameter so that the DSA parameter *r* is a quadratic nonresidue modulo *P*. To communicate the subliminal message "1", the subliminal transmitter chooses *k* so that *r* is a quadratic residue modulo *P*.

For a positive integer *A* less than *P*, *A* is a quadratic residue modulo *P* if

$$X^2 \equiv A \text{ (modulo P)} \qquad (18)$$

for some *X*. A prime *P* has *(P - 1)/2* quadratic residues modulo *P* and *(P - 1)/2* quadratic nonresidues modulo *P*. For example, if *P* is 7, then the quadratic

residues are 1, 2, and 4, as shown below. The quadratic nonresidues are 3, 5, and 6 because there are no values of X that satisfy the above expression.

$1^2 \equiv 1$ (modulo 7)
$2^2 \equiv 4$ (modulo 7)
$3^2 \equiv 2$ (modulo 7)
$4^2 \equiv 2$ (modulo 7)
$5^2 \equiv 4$ (modulo 7)
$6^2 \equiv 1$ (modulo 7)

To extend this subliminal channel in DSA so that multiple subliminal bits are sent in a single signed message, the subliminal transmitter and the subliminal receiver agree on multiple shared secret keys. For example, to send two subliminal bits per message, primes P and Q are established. A value of k is chosen so that r is either a quadratic residue modulo P or a quadratic nonresidue modulo P, and that r is either a quadratic residue modulo Q or a quadratic nonresidue modulo Q.

It is possible that a malicious implementation of DSA can leak bits of the signer's private key [SCHN96]. As long as the primes (P, Q, etc.) chosen by the implementation stay a secret, the signer can not prove that his private key was stolen.

These subliminal channels in DSA do not work if the k parameter can not be chosen arbitrarily. One technique proposed [SIMM94a] is to have the sender and receiver jointly choose k. For example,

$$k = k' k'' \text{ (modulo } (p-1)) \qquad (19)$$

where k' is selected by the sender, k'' is selected by the receiver, and p is the DSA parameter. Simmons observed [SIMM94a] that this technique can allow the receiver to embed a subliminal message in the sender's signature by choosing a particular k'' value. Simmons dubbed this the "Cuckoo's Channel."

Other Types of Hidden Channels

There are many other possible types of hidden channels beside subliminal channels in digital signatures. For example, many cryptographic protocols use random values called nonces. However, as we saw in one subliminal channel in DSA, the 160-bit *k* parameter was also supposed to be a random number. Recall that although *k* was used to convey the secret message, *k* was made to look random by XORing the secret message with a one-time pad. A similar thing could be done with nonces.

Recall that timestamps are frequently used in cryptographic protocols; one purpose of a timestamp is to frustrate a replay attack. As in the steganographic technique for hiding a secret message in a digital image file, the least significant units of time (for example, the milliseconds place) could be modified to communicate a secret message. Particularly because computer system clocks are rarely in synchronization to the precision of milliseconds, this modification would be difficult to detect.

There might also be hidden channels available below the application layer. Information can also be passed from system to system by exploiting the communication channel itself.

A good example is found in the TCP/IP protocol suite [ROWL2005]. Normal-looking packets can carry secret information if we place this information in certain fields of the TCP/IP header. Although the optional fields may be used for this purpose, required fields are preferable because they are less likely to be altered during transmission.

One required field that may be manipulated is the IP Identification field. The IP Identification field was designed to contain a unique value so that if a packet is fragmented during transmission, it could be correctly re-assembled at the receiver.

However, because there were no other requirements placed on the IP Identification field, this 16-bit field is available for use in covert transmissions.

Another interesting field is the 32-bit TCP Initial Sequence Number field. Clearly, an available 32 bits per packet provides more flexibility than 16 bits per packet. However, because publicly available tools such as Ethereal (www.ethereal.com) can detect out-of-order sequence numbers for a TCP session, this may be a less-desirable technique unless multiple short-lived sessions are used.

Applications and Implications

One frequently cited use of a subliminal channel is the situation where a message is signed under threat. The signer agrees to sign the message, but imbeds the subliminal message that he was coerced. Another possible application involves marking files or digital cash to allow tracking by the entity that produced the mark. Malicious software such as spyware could be used to leak secret information from a host computer without the leak being detected.

It is important to note that the existence of a subliminal channel could compromise the security of a system. Techniques for creating subliminal channels should be studied, in order to understand possible threats and avenues of attack.

Block Encryption with Multiple-Key Sets

In this section, we investigate extending common symmetric key block cipher modes of operation by using multiple symmetric keys. Unlike traditional multiple-key encryption techniques, each block of plaintext is encrypted using a single key.

Concept

The sender and receiver generate and share a set of symmetric keys. Every key in the set is generated independently of the others. For simplicity, we begin by presenting the case where there are only two keys in the set. Denote these keys *Ks1* and *Ks2*.

$$\text{Set} = \{\text{Ks1}, \text{Ks2}\} \tag{20}$$

The key used to encrypt each plaintext block is selected by a sequence. Each entry in the sequence chooses one key from the set, using a predetermined mapping. For example,

Sequence entry:	Encrypt with key:
0	Ks1
1	Ks2

Each block of the plaintext message is encrypted with either *Ks1* or *Ks2*. As shown in the mapping table, if the sequence entry is a 0 bit, then the block is encrypted with *Ks1*. If the sequence entry is a 1 bit, then the block is encrypted with *Ks2*.

If each key in the set is generated independently, and each key is sufficiently strong on its own, then this technique should not weaken the encryption of the message.

Clearly, it is possible to use more than two keys in the set. If both the sender and the receiver know the sequence, then it is a straightforward extension of the technique to use twenty keys rather than two keys. (Of course, there are costs associated with using additional keys. These costs will be discussed later.) If more than two keys are used, then sequence entries can no longer be one-bit binary values. Consider the following example for five keys in the set.

Sequence entry (version 1):	Sequence entry (version 2):	Sequence entry (version 3):	Sequence entry (version 4):	Encrypt with key:
0	000	00 (H)	Buy	Ks1
1	001	01 (H)	Sell	Ks2
2	010	02 (H)	Hold	Ks3
3	011	03 (H)	Upgrade	Ks4
4	100	04 (H)	Downgrade	Ks5

A plaintext message *M* can be represented as

$$M = M1, M2, M3, ..., Mk \quad (21)$$

Each *Mi* is a block of length *L*, and *M* consists of *k* blocks. Let the sequence be represented as

$$b = b1, b2, b3, ..., bk. \quad (22)$$

Each *bi* is a single entry in the sequence. In the simple binary case, each *bi* could assume the value 0 or 1. Let the key selected using entry *bi* be denoted *Ks(bi)*. The resulting ciphertext *C* can be represented as

$$C = C1, C2, C3, ..., Ck \quad (23)$$

The ciphertext blocks are presented in the same order as the corresponding plaintext blocks. That is, *Ci* is the result of encrypting *Mi* using the key *Ks(bi)* selected by *bi*.

ECB Mode with a Multiple-Key Set (MK-ECB)

MK-ECB mode encrypts each block independently of the other blocks in the message. Equation 24 and Figure 9 describe this encryption technique.

$$Ci = Encrypt [Ks(bi), Mi] \quad (24)$$

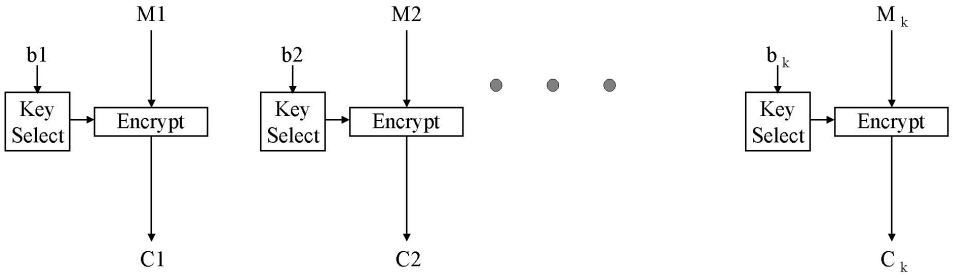

Figure 9: Encryption in MK-ECB Mode

If the sequence is known to the receiver, then the decryption technique is similar to the encryption technique, as described by Equation 25 and Figure 10. The case where the sequence is not known to the receiver is described later, in the applications section.

$$Mi = Decrypt [Ks(bi), Ci] \quad (25)$$

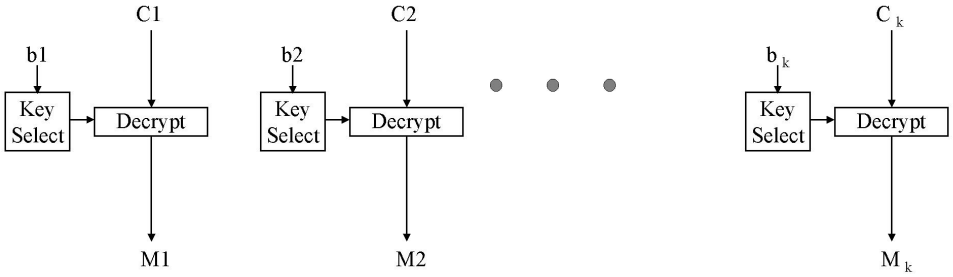

Figure 10. Decryption in MK-ECB Mode (Receiver knows the sequence)

Because there is a direct correlation between each block of plaintext and its ciphertext, an attacker could exploit patterns that occur in the plaintext. Therefore, like ECB mode, MK-ECB mode is recommended for short messages only.

It is possible that, for example, if a plaintext block is repeated, each of the duplicate plaintext blocks could be encrypted with a different key. A sequence chosen at random might accomplish this automatically. However, this could be a risky proposition. With only two keys in a set, the probability of selecting a given key at random is 50%.

CTR Mode with a Multiple-Key Set (MK-CTR)

MK-CTR mode, like CTR mode, encrypts each block independently of the other blocks in the message. Equation 26 and Figure 11 describe this encryption technique. The initial counter value is represented by *counter*. Recall that the symbol ⊕ represents an XOR operation.

$$Vi = \text{Encrypt}\,[Ks(bi), \text{counter}+(i-1)] \qquad (26)$$
$$Ci = Vi \oplus Mi$$

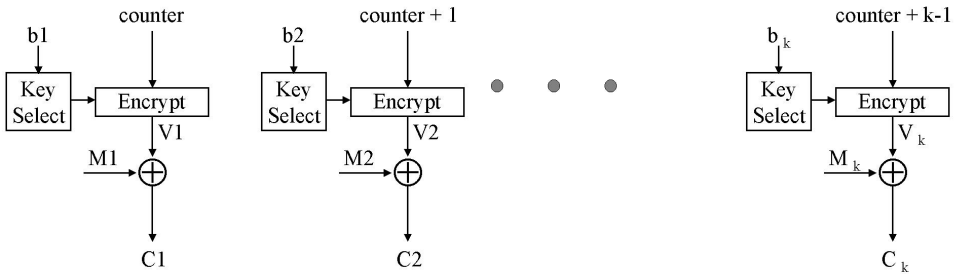

Figure 11: Encryption in MK-CTR Mode

If the sequence is known to the receiver, then the decryption technique is similar to the encryption technique, as described by Equation 27 and Figure 12. The case where the sequence is not known to the receiver is described later, in the applications section.

$$Vi = \text{Encrypt}\,[Ks(bi), ctr+(i-1)] \qquad (27)$$
$$Mi = Vi \oplus Ci$$

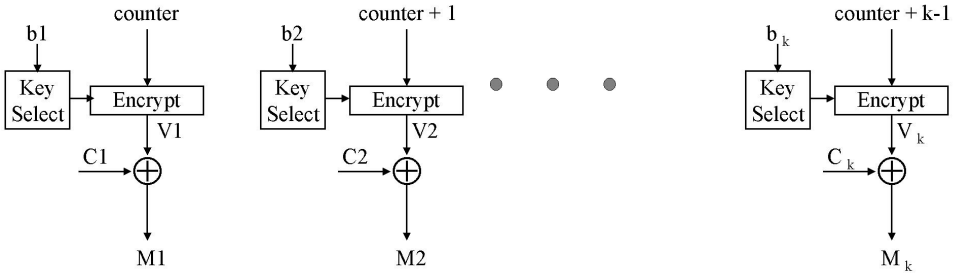

Figure 12. Decryption in MK-CTR Mode (Receiver knows the sequence)

Recall that in CTR mode, unlike ECB mode, even if the same plaintext block is repeated, the corresponding ciphertext blocks will be different. Also, recall that the intermediate values, here labeled *Vi*, may be pre-computed. These properties hold true for MK-CTR. In general, MK-CTR is preferred over MK-ECB.

$$Vi = \text{Encrypt}\,[Ks(bi), ctr+(i-1)] \qquad (28)$$

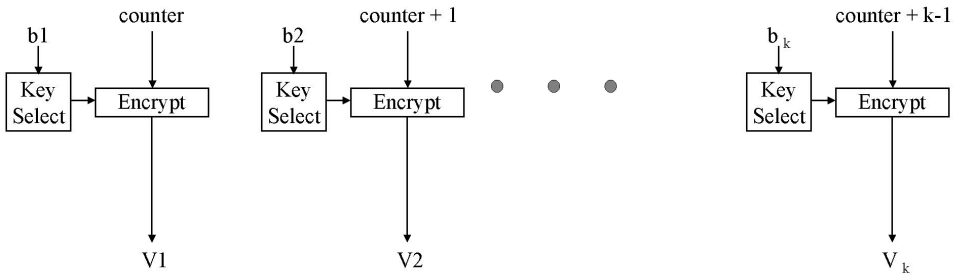

Figure 13: MK-CTR Mode for Generation of Intermediate Values

A variation of MK-CTR mode, used only to generate intermediate values, is described by Equation 28 and Figure 13. A use for these intermediate values is discussed in the applications section.

CBC Mode with a Multiple-Key Set (MK-CBC)

In MK-CBC mode, as in CBC mode, each ciphertext block is a function of all the previous plaintext blocks in the message. Therefore, encryption is sequential. Equation 29 and Figure 14 describe this technique.

$$C1 = \text{Encrypt}\,[Ks(bi), (M1 \oplus IV)] \qquad (29)$$
$$Ci = \text{Encrypt}\,[Ks(bi), (Mi \oplus C_{i-1})] \quad (\text{for } k \geq i \geq 2)$$

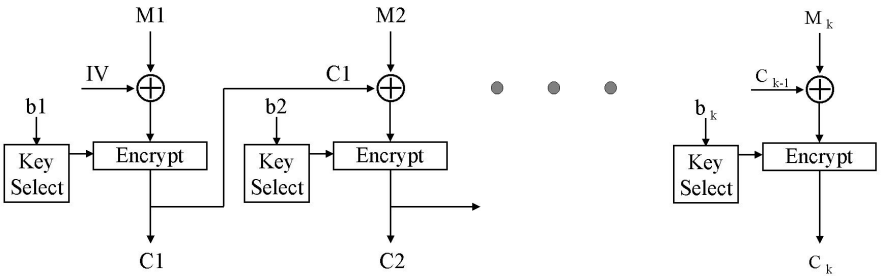

Figure 14: Encryption in MK-CBC Mode

If the sequence is known to the receiver, then the decryption technique is the reverse of the encryption technique, as described by Equation 30 and Figure 15.

$$M1 = \text{Decrypt}\,[Ks(bi), C1] \oplus IV \qquad (30)$$
$$Mi = \text{Decrypt}\,[Ks(bi), Ci] \oplus C_{i-1} \qquad (\text{for } k \geq i \geq 2)$$

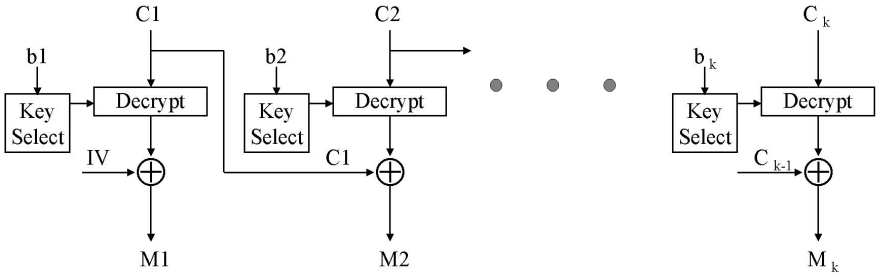

Figure 15. Decryption in MK-CBC Mode (Receiver knows the sequence)

In MK-CBC mode, as in CBC mode, even if a plaintext block is repeated, the corresponding ciphertext blocks will be different. The last block produced in MK-CBC mode (C_k) is a function of all the plaintext blocks in the message and it is a function of all the keys selected to encrypt. In other words, C_k is a function of the sequence.

CBC-MAC with a Multiple-Key Set (MK-CBC-MAC)

Recall that traditional CBC-MAC uses CBC mode with an *IV* of zero. The same encryption key is used for every block. The final block of the CBC output (*Ck*) is the MAC. MK-CBC-MAC is similar to CBC-MAC, except that the sequence selects each key in the chain.

Suppose there are n independent keys in a set, each key is of length L, the encryption block size is B bits, and the sequence is k entries in length. We assume that the sequence is chosen at random. There are n^k different sequences possible; therefore, an attacker has an $n^{(-k)}$ chance of correctly guessing the sequence.

Further, suppose the attacker knows m keys in the set, where m is strictly less than n. Assuming the sequence selects each of the n keys at least once, the attacker's job is at least as difficult as finding the $(n-m)$ keys he is missing.

Now suppose all n keys in the set are known to an attacker, but the attacker does not know the sequence. If the MAC and message are not encrypted, then the attacker knows C_k. However, the attacker does not know C_1, C_2, ... C_{k-1}. Therefore, it is impossible for the attacker to simply encrypt each plaintext block with each of the n keys in the set to determine which key was used to produce the corresponding ciphertext block. Instead, the attacker must guess up to n^k sequences, each of length k. This means that in a brute force attempt to guess the sequence, each of the n^k trials requires k sequential encryption computations.

For example, let $n = 2$ and $k = 100$, so the set contains two keys and the message has 100 blocks. Note that this is a relatively short message. If each block is 128 bits ($B = 128$), then the size of the message is $B \times k = 12800$ bits = 1600 bytes = 1.6 KB. However, even for this short message, there are $2^{100} \approx 10^{30}$different sequences possible. Even if the attacker knows both of the $n = 2$ keys, the attacker may have to perform as many as 2^{100} trials, with each trial requiring 100 sequential

encryption computations. Clearly, a brute force attack against MK-CBC-MAC is inefficient.

Other Modes with a Multiple-Key Set

Other common modes of operation include Cipher Feedback (CFB) and Output Feedback (OFB). It is possible to implement MK-CFB and MK-OFB. However, we will not discuss these modes in detail.

Comments on Efficiency

For block ciphers such as AES and DES, the encryption key must be scheduled before it is used to encrypt the plaintext, or before it is used to decrypt the ciphertext. The key scheduling algorithm can be complex, and it may take some time to complete. When a message is encrypted using different keys for each block, the key scheduling algorithm would have to be run frequently.

However, if we "demux" the blocks in the sequence, then we could reduce the number of times we run the key scheduling algorithm. We could schedule key *Ki*, then encrypt (or decrypt) all the blocks the sequence indicates should be encrypted (or decrypted) with *Ki*. When we are finished with *Ki*, we schedule key *Kj*, and so on. Once we have processed all the blocks in the message, we "mux" the blocks back into the correct order.

This is similar to the technique of interleaving. In fact, if multiple instantiations of encryption hardware are available, each hardware instantiation could handle encrypting the plaintext blocks for a given key.

Note that this technique of "demuxing" the blocks in the sequence works as expected for MK-ECB and MK-CTR modes. However, for modes where

encrypting the current block depends on the encryption result from the previous block, such as MK-CBC, this technique produces different ciphertext.

For example, let the set contain two keys, and let the first key be mapped to sequence entry 0 and the second key be mapped to sequence entry 1. Let the sequence $b = 0, 1, 1, 0, 1$. This implies that blocks 1 and 4 are encrypted using the first key, and blocks 2, 3, and 5 are encrypted using the second key. Without "demuxing" the blocks, the resulting ciphertext blocks are:

C1 = Encrypt [Ks(0), (M1 ⊕ IV)]
C2 = Encrypt [Ks(1), (M2 ⊕ C1)]
C3 = Encrypt [Ks(1), (M3 ⊕ C2)]
C4 = Encrypt [Ks(0), (M4 ⊕ C3)]
C5 = Encrypt [Ks(1), (M5 ⊕ C4)] (C1, C2, C3, C4, C5 form a chain)

With "demuxing" the blocks, the resulting ciphertext blocks are:

C1' = Encrypt [Ks(0), (M1 ⊕ IV1)]
C4' = Encrypt [Ks(0), (M4 ⊕ C1)] (C1 and C4 form a chain)
C2' = Encrypt [Ks(1), (M2 ⊕ IV2)]
C3' = Encrypt [Ks(1), (M3 ⊕ C2)]
C5' = Encrypt [Ks(1), (M5 ⊕ C3)] (C2, C3, C5 form a chain)

Notice that two different *IV* values must be used, one for each chain. A chain of blocks is now defined by the blocks that are encrypted with a given key. Observe that all ciphertext blocks except *C1* are different (e.x., $C2' \neq C2$).

Possible Applications

Possible applications of the techniques include establishing a covert channel, customizing message encryption at the block level, and expanding a string of pseudo-random numbers.

Establishing a Subliminal Channel

Subliminal channels were described in the background section. Much of the published work relates to hidden communication channels in digital signatures, or messages hidden within the communications protocol itself.

Recall that in the Prisoner's Problem scenario, the sender and receiver must communicate in the open. They use an authentication without secrecy channel, and send hidden messages by subverting the authentication mechanism.

Here, we assume that the sender and receiver are allowed to have secrecy. They may encrypt messages before transmitting on the channel. The encryption scheme is extended to include an additional message hidden in each block of ciphertext.

A common theme among subliminal channels is that the sender is permitted to make a choice between different values. This choice communicates information to the receiver. For example, recall that in the proposed subliminal channel for DSA, the sender chooses the k parameter based on whether it produces an r value that is a quadratic nonresidue modulo a prime P.

Here, the sender chooses which key will encrypt each block of plaintext. Recall that the sequence entry bi selects one key from the set of shared symmetric keys, and this key is used to encrypt block i. The message hidden in the block is the sequence entry corresponding to that block. That is, for block i the sequence entry bi represents the hidden message.

Note that the entry *bi* could be a binary digit, a binary *n*-tuple, a decimal or hexadecimal number, an ASCII character, and so on. The entry *bi* could even be a pointer to a file, an Internet address, or a predetermined codeword with a specified meaning. The entire sequence could represent a hidden message, so that the particular ordering of the entries in the sequence conveys meaning to the intended receiver.

For the receiver to recover the hidden message (and the original plaintext message), the receiver must be able to determine which key in the set was used to encrypt the block. If the set of keys is relatively small, then the receiver can guess one of the keys, decrypt the ciphertext block using that key, and check whether the recovered plaintext is correct. Note that if the set of keys is very large, then the receiver's job begins to resemble a brute force attack on the key.

It is possible that both the sender and the receiver know the original plaintext message prior to the exchange. For example, the receiver may be able to predict the contents of certain communication protocol messages, such as header values. Thus, the sender uses this technique only on those messages that it knows the receiver can predict. Alternatively, the sender and the receiver may agree on a plaintext message prior to the communication, such as the text of the novel War and Peace, or a predetermined (and lengthy!) one-time pad.

However, if the receiver does not know the original plaintext block, then the receiver needs to be able to determine whether the recovered plaintext block is correct, based only on the recovered plaintext block itself. For example, the original plaintext block could include a known type of formatting, such as Tag-Length-Value formatting, specified by [ASN2002]. Alternatively, each *Mi* could contain a rare symbol or combination of symbols that act as a signal for the receiver. For example, such a signal could be one ASCII "bell" symbol, one ASCII "acknowledge" symbol, and a second ASCII "bell" symbol: 07 06 07 (hexadecimal representation).

Another alternative is to use a function of the plaintext bits to create a signal for the receiver. For example, an error-detecting code could be included in each block before encryption. This increases the probability that the signal to the receiver is different for each block, but also increases the processing work for each block of the message.

Finally, the software application could require end user involvement to determine whether the recovered plaintext block is correct. This last technique may actually be most desirable in certain situations. Particularly if the plaintext is human-readable, this technique could be performed quickly by a human user, with no need to add extra signaling bits to each plaintext block. Even if the plaintext is not a text message, a signal could be embedded in the plaintext that a human could recognize. The signaling technique could be changed frequently, without any reliance on a software or hardware implementation. The signaling technique could be memorized by the sender and the receiver, so that the technique does not even need to be written down. This allows the sender and receiver to apply any steganographic technique they can imagine to generate a signal indicating the block is correct. (For example, imagine the digital equivalent of microscopic dots.)

A covert channel can be established using the block cipher modes of operation described in the block encryption with multiple-key sets section. The covert channel could be used in a straightforward manner to convey a hidden message. It could also artificially expand the capacity of a communication channel, or reduce the overhead required to periodically update session keys.

Hiding a Message

To convey a hidden message, the sender chooses a technique to signal the receiver that each block is correct, adds any necessary bits to the blocks, constructs the sequence, and uses the sequence to encrypt the plaintext blocks.

The receiver knows the keys in the set, and the mapping between each key and its sequence entry. (These have been agreed upon in advance.) However, the receiver does not know the particular sequence used to encrypt the plaintext blocks. When the receiver obtains a ciphertext block *Ci*, the receiver chooses a key from the set and decrypts the block to obtain *Mi**. The receiver tests whether *Mi** is correct. If *Mi** is correct, then *Mi** = *Mi*, and the sequence entry *bi* is known. If *Mi** is not correct, then the receiver chooses a different key from the set and tries again. The process repeats until the receiver obtains a correct *Mi** = *Mi* and the sequence entry *bi* is known.

To illustrate this process, we continue our simple example. The set contains keys *Ks1* and *Ks2*, where *Ks1* is mapped to 0 and *Ks2* is mapped to 1. The decryption technique is described below.

Mi* = Decrypt [Ks1, Ci]
Test if Mi* is correct
    If Mi* is correct, then bi=0 and Mi = Mi*
    If Mi* is not correct, then bi=1 and Mi = Decrypt [Ks2, Ci]

That is, if decryption is successful and *Mi** is correct, then *Mi* = *Mi** and *bi* = 0. If *Mi** is not correct, then *Mi* ≠ *Mi** and *bi* = 1. To recover *Mi*, the receiver repeats the decryption step, this time using *bi* = 1 instead of *bi* = 0.

Figure 16 illustrates the process in MK-ECB mode, Figure 17 illustrates the process in MK-CTR mode, and Figure 18 illustrates the process in MK-CBC mode.

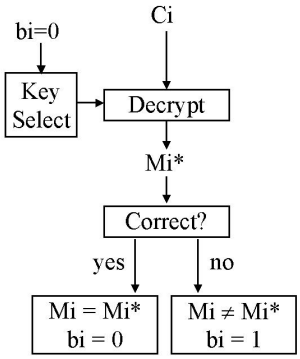

Figure 16. Decryption in MK-ECB Mode (Sequence Unknown)

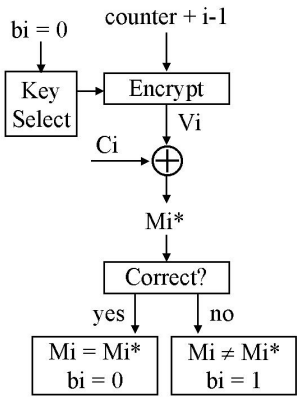

Figure 17. Decryption in MK-CTR Mode (Sequence Unknown)

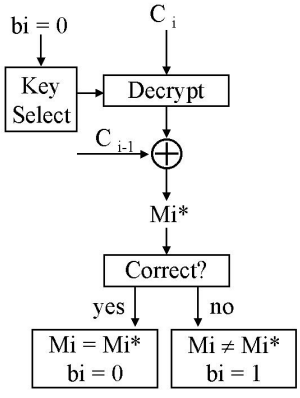

Figure 18. Decryption in MK-CBC Mode (Sequence Unknown)

Note that decryption in MK-CBC mode requires that the receiver know both the current ciphertext block (C_i) and the previous ciphertext block (C_{i-1}). In contrast, the receiver only needs to know the current ciphertext block (C_i) to decrypt in MK-ECB mode. In MK-CTR mode, the receiver needs to know the current ciphertext block (C_i) and the current counter value (*counter+i-1*). Note that in MK-CTR mode, both *Vi* values (that is, one *Vi* value for *Ks1* and one *Vi* value for *Ks2*) can be pre-computed by the receiver before *Ci* arrives. This could significantly speed up the "decryption" step performed by the receiver.

Decrypt Requirements

In our simple example where the set only contains two keys, the receiver only needs to decrypt once to recover the sequence entry *bi* for the block. After the block is decrypted, the test reveals that either *bi*=0 or *bi*=1. However, the average number of tries needed to recover the original plaintext block *Mi* is described by Equation 31. We assume that each sequence entry is equally likely. Therefore, on average, the correct key is tried fifty percent of the time, and the incorrect key is tried fifty percent of the time.

$$\begin{aligned}(50\%)(\text{tried correct key}) + (50\%)(\text{tried incorrect key}) \qquad (31)\\ = (50\%)(\text{takes one try}) + (50\%)(\text{takes two tries})\\ = (0.50)(1) + (0.50)(2)\\ = 1.50 \text{ tries}\end{aligned}$$

If a set contains more than two keys, then the receiver will need to perform more decryption steps to recover *bi* and *Mi*. Let *n* be the number of keys in the set. To recover *Mi*, the receiver tries keys from the set until it finds a key the produces the correct plaintext. For the general case of *n* keys in the set, the average number of tries needed to recover the original plaintext block *Mi* is described by Equation 32.

$$\text{Average number of tries to recover Mi} = \frac{\sum_{i=1}^{n} i}{n} \qquad (32)$$

For the general case of *n* keys in the set, the average number of tries needed to recover the sequence entry *bi* for the block is described by Equation 33.

$$\text{Average number of tries to recover bi} = \frac{(\sum_{i=1}^{n-1} i)+(n-1)}{n} \qquad (33)$$

The number or tries to recover *bi* and *Mi*, for several values of *n*, is presented below.

n	Average tries for bi	Average tries for Mi
2	1	1.5
4	2.25	2.5
8	4.375	4.5
16	8.4375	8.5

Comments on Efficiency

Recall that, for block ciphers such as AES and DES, the key must be scheduled before it is used to decrypt the ciphertext. Because the key scheduling algorithm can be complex and take time to complete, we would like to reduce the number of times we run the key scheduling algorithm.

In a previous section, we described a technique of "demuxing" and "muxing" the blocks in the sequence. Here, we describe a different technique.

The receiver has *k* blocks of ciphertext to decrypt. He schedules key *Ki*, then decrypts all of the blocks using *Ki*. He checks to see which blocks are correct. For each correct block, the receiver knows the recovered plaintext and the sequence entry. The recovered plaintext and the sequence entry are stored, and the incorrect blocks remain. The receiver schedules key *Kj*, then decrypts all of the remaining ciphertext blocks using *Kj*. This process continues until all the blocks in the message are correct. For *n* keys in the set, the receiver must run the key scheduling algorithm at most *n* times.

Practicality

Why would anyone want to use this technique? After all, the sender and receiver can already encrypt the messages they exchange using a strong cryptographic algorithm.

Suppose one of the keys, say *Ks1*, in the simple two-key set is stolen. The attacker can learn the pattern of the keys used to encrypt the original plaintext message. However, the attacker may not be able to determine the mapping of the keys in the set to the sequence entries (assuming the mapping is a little more creative than the mapping in our simple example). In addition, the attacker can only recover about half of the original plaintext message.

Suppose one of the keys in a 20-key set is stolen. The attacker can discover about $1/20^{th}$ of the pattern of the keys, and the attacker can recover about $1/20^{th}$ of the original plaintext message.

Thus, a degree of fault-tolerance is obtained by using multiple keys in this manner. The tradeoff is that it is necessary to exchange and store more keys, and perform additional processing for each block of the message. In certain situations, this additional work may be worthwhile.

Channel Capacity

It may also be possible to utilize this technique to artificially increase the capacity of a communications channel. Assume we neglect any overhead (bits added to each plaintext block) required to determine whether a recovered plaintext block is correct. We can consider the capacity of the channel to have increased by an amount related to the size of the set.

Consider the AES block size of 128 bits per block. If a two-key set is used, with each key mapped to a single binary bit, either a '0' or a '1', then the sender conveys one extra bit of information per block. Therefore, 129 bits of information is conveyed for every 128 bits sent, for a gain of 129/128 = 1.0078125 = 0.78%.

If a *n*-key set is used, with each key mapped to a single decimal number between 1 and *n*, then the sender could convey $\log_2(n)$ bits of information with every 128 bits sent. For example, let *n* be 256, which indicates that there are 256 independent keys in the set. One of these 256 keys is selected to encrypt each plaintext block. For example, say key number 143 is selected to encrypt a block. The representation of 142 in binary is 10001110. We observe that there are $\log_2(256)$ = 8 bits of additional information conveyed per block. This scheme provides a capacity gain of 136/128 = 1.0625 = 6.25%.

It is possible that even more information could be conveyed by the sender in each block, if the sender is creative about mapping the keys in the set. (As before, this delves into the realm of steganography.)

Again, we consider the practicality of this technique. Consider the extreme example of a communications channel between the Earth and Mars. The cost of sending a single bit of information likely outweighs the cost of processing plus the cost of storing additional keys. Therefore, in certain situations, any added capacity may be worth the extra work.

Updating Session Keys

In many communications protocols, it is necessary to periodically update an old symmetric key (also called a session key) with a new symmetric key. This process is called re-keying.

The technique described above could be used to transmit raw key material between the sender and the receiver while encrypted messages are exchanged. This allows the session key to be updated without the overhead of exchanging additional protocol messages solely for re-keying purposes. In other words, the exchange of messages is never interrupted by re-keying protocol messages.

An example of this type of covert re-keying protocol is described below. The sender and the receiver begin the protocol sharing initial symmetric keys *Ks1* and *Ks2*, a key mapping to sequence entries, and a master key *Km*. The master key *Km* is the same size as *Ks1* and *Ks2*, and does not change when *Ks1* and *Ks2* are updated. For simplicity, we describe *Km* as a symmetric key; however, pairs of asymmetric keys could be used instead.

The purpose of *Km* is to provide forward secrecy. Without forward secrecy, if an attacker discovers *Ks1* or *Ks2*, then the attacker can determine the covertly exchanged sequence *b*. If the sequence *b* is used without modification to form the new keys *Ks1'* (the new *Ks1*) and *Ks2'* (the new *Ks2*), then the attacker uses his knowledge of the old keys to determine the new keys. In fact, without forward secrecy, the attacker could determine all keys created after *Ks1* and *Ks2*.

In the following protocol, if *Ks1* and *Ks2* are discovered by an attacker, then we want to ensure that *Ks1'* and *Ks2'* are not revealed to the attacker. We do this by encrypting the raw key material contained in the sequence *b* with *Km*.

Initial State: Sender and Receiver share: *Km*, *Ks1*, *Ks2*, and mapping of keys to sequence entries (for example, *Ks1* to 0 and *Ks2* to 1)

Loop

- Sender (has original plaintext message M_*sender* to send):
  1. creates sequence *b_sender* (sender's raw key material to contribute)
  2. generates *C_sender* using MK-CTR or MK-CBC mode (encodes *b_sender*)
  3. sends *C_sender* to Receiver
- Receiver:
  1. receives *C_sender*
  2. recovers *M_sender* and *b_sender* using same mode
- Receiver (has original plaintext message *M_receiver* to send):
  1. creates sequence *b_receiver* (receiver's raw key material to contribute)
  2. generates *C_receiver* using same mode (encodes *b_receiver*)
  3. sends *C_receiver* to Sender
- Sender:
  1. receives *C_receiver*
  2. recovers *M_receiver* and *b_receiver* using same mode
- Sender and Receiver both (after sending *k* bits of raw key material and receiving *k* bits of raw key material):
  1. Interleave *b_sender* and *b_receiver* to form *b*:
     $b = b\_sender_1, b\_receiver_1, b\_sender_2, b\_receiver_2, \ldots b\_sender_k, b\_receiver_k$
  2. split *b* into two halves, *bx* and *by*:
     $b = bx \mid by$
  3. build the new *Ks1* and new *Ks2*:
     Ks1' = Encrypt [*Km*, *bx*]
     Ks2' = Encrypt [*Km*, *by*]
- Sender (when finished building *Ks1'* and *Ks2'*): uses the covert channel to signal that this is the last block the sender will encrypt with the old keys *Ks1* and *Ks2*, and that Receiver should expect the next block received to be encrypted with *Ks1'* and *Ks2'*
- Receiver (when finished building *Ks1'* and *Ks2'*): uses the covert channel to signal that this is the last block the receiver will encrypt with the old keys *Ks1* and *Ks2*, and that Sender should expect the next block received to be encrypted with *Ks1'* and *Ks2'*

The above protocol works best if the communication is reasonably symmetrical. That is, the Sender and the Receiver both send about the same amount of information over the interval of time that it takes to re-key. If the communication is not reasonably symmetrical, then the Sender and the Receiver will have to wait until both have sent enough raw key material to re-key. This implies that while

they are waiting, they must store the raw key material that they have already sent and received. This storage space could add up over the long term.

Consider the following example with k=128, where the Sender sends twice as many blocks as the Receiver during a given time interval.

Sender: *C1_sender*, *C2_sender*, ... *C94_sender*	(94 blocks)
Receiver: *C1_receiver*, *C2_receiver*, ... *C29_receiver*	(29 blocks)
Receiver: *C30_receiver*, ... *C42_receiver*	(13 blocks)
Sender: *C95_sender*, ... *C156_sender*	(62 blocks)

At this point, the Sender has sent more than 128 bits of raw key material. The Receiver has only sent 42 bits of raw key material. New keys cannot be generated yet because we have to wait for the Receiver to send at least 128 bits of raw key material.

Receiver: *C43_receiver*, ... *C64_receiver*	(22 blocks)
Receiver: *C65_receiver*, ... *C91_receiver*	(27 blocks)
Sender: *C157_sender*, ... *C221_sender*	(65 blocks)
Receiver: *C92_receiver*, ... *C119_receiver*	(28 blocks)
Sender: *C222_sender*, ... *C283_sender*	(62 blocks)
Receiver: *C120_receiver*, ... *C144_receiver*	(25 blocks)

The Receiver has finally sent more than 128 bits of raw key material. New keys can now be generated. Note that the Sender has sent more than 256 bits of raw key material, which is enough for its contribution to two new keys.

In a variation to the above protocol, only the Sender supplies raw key material. Here, the Sender must supply enough raw key material to create both *Ks1'* and *Ks2'* on its own. This variation works best if the Sender transmits the bulk of the information. However, if the Sender is malicious, then the Sender could compose weak keys on purpose since it supplies all the raw key material. If the Receiver discovers that the Sender is composing weak keys, then the Receiver's only choice is to permit the use of weak keys or terminate communication with the Sender.

Customizing Message Encryption at the Block Level

Message encryption can be customized at the block level using the block cipher modes of operation described in the block encryption with multiple-key sets section.

With the ability to customize the encryption of a message at the block level, several types of applications are possible. One possible application is managing access control for different portions of a document. Another possible application is keeping the initiator of a contest (or protocol) honest. We explore these possible applications later in this section.

In these applications, we assume the sequence is known to the receivers. (Optionally, the sequence may be unknown to the receivers, but this only forces the receivers to work harder.) Each receiver has certain keys, and uses those keys to try to decrypt the message. Each key is mapped to a particular sequence entry. The mapping is known to the sender and to the receivers. The sender chooses a sequence and then uses MK-ECB mode, MK-CTR mode, or MK-CBC mode to encrypt the original plaintext. Each receiver then uses the same mode to decrypt the ciphertext.

Managing Access Control for Portions of a Document

Suppose we have a group of individuals who need to work together. The group consists of managers and engineers from Company A, managers and engineers from Company B, researchers at University C, and employees of Government Agency D. Each subgroup has information to contribute to a project. However, each subgroup also has information that they do not want the others to see. For example, C might authorize D to see information that he does not want anyone

from A or B to see. A might grant authorization to view certain information to all members of the group except B, since B is a direct competitor of A. This is a common occurrence in working groups of standards organizations, and other collaborative efforts.

Some current applications use back-end servers to customize page views, based on authorization levels. This kind of application can manage which users can access information, if the information stays on the centralized server.

Instead, suppose that the members of a group want to send and receive files directly, without the assistance of an online application. Each document author still needs to manage who is permitted to see particular information. The author could generate multiple versions of the document and send each version separately. Alternatively, the author could generate a single copy of the document, but encrypt portions of the document using different keys. Suppose one key is shared by all members of the group, and other keys are shared only by subgroups. For example:

Key	Shared by
1	All members of the group
2	Managers from A, Engineers from A
3	Managers from A
4	Managers from B, Engineers from B
5	Managers from B
6	C only
7	D only
8	A and B only
9	A (Managers only) and B (Managers only) only
10	A (Managers only) and C only
11	A (Managers only) and D only
12	B (Managers only) and C only
13	B (Managers only) and D only
14	C and D only
15	A (Managers only) and B (Managers only) and C only
16	A (Managers only) and B (Managers only) and D only
17	A (Managers only) and C and D only
18	B (Managers only) and C and D only

Each subgroup may have a different number of keys. In this example, a manager at A stores ten keys: 1, 2, 3, 8, 9, 10, 11, 15, 16, and 17. An engineer at A stores two keys: 1 and 2. An employee at C stores eight keys: 1, 6, 10, 12, 14, 15, 17, and 18. (Recall that techniques exist that may reduce the number of keys that must be exchanged, such as the variant of asymmetric key cryptography described in [BOYD88] and presented in the background section. However, because our focus is symmetric key block ciphers, in this example we use the basic key distribution described above.)

Now that the keys have been generated, exchanged and stored, each author can customize the permissions on portions of the document by choosing which keys will encrypt which blocks. The author only needs to update the sequence *b* associated with the document.

Suppose an engineer at A writes a technical document and encrypts the entire document with *Key2*. All the managers and engineers at A can read the document. However, no one outside this subgroup (defined as those individuals who share *Key2*) can read the document, even if the document is sent to the entire group by mistake. Manager 1 at A adds some comments to the document that he encrypts with *Key3*. Manager 2 at A replies to Manager 1's comments, again encrypting with *Key3*. The Managers at A decide to send portions of the document to C, so they decrypt only those portions and re-encrypt them with *Key10*. When the researchers at C receive the document, they can only read the portions that were encrypted with *Key10*. They decrypt these portions, add new information, and re-encrypt with *Key1*. All members of the group see the result.

For example,

> Text1a: "I have discovered a new way to ... The confidential part is ..."
> Text1b: "Here is the part that is not confidential ..."
> Text2: "I like it. Let's share the part that is not confidential."
> Text3: "Agreed. We'll share only that part."
> Text4: "Here is a new idea our engineers discovered."
> Text5: "Interesting idea. What if we also did ..."

The documents created are:

> Document1 = Encrypt [Key2, Text1a] || Encrypt [Key2, Text1b]
> Document2 = Encrypt [Key3, Text2] || Encrypt [Key2, Text1a] ||
> Encrypt [Key2, Text1b]
> Document3 = Encrypt [Key3, Text3] || Encrypt [Key3, Text2] ||
> Encrypt [Key2, Text1a] || Encrypt [Key2, Text1b]
> Document4 = Encrypt [Key10, Text4] || Encrypt [Key3, Text3] ||
> Encrypt [Key3, Text2] || Encrypt [Key2, Text1a] ||
> Encrypt [Key10, Text1b]
> Document5 = Encrypt [Key1, Text5] || Encrypt [Key1, Text4] ||
> Encrypt [Key3, Text3] || Encrypt [Key3, Text2] ||
> Encrypt [Key2, Text1a] || Encrypt [Key1, Text1b]

Notice that the text itself is not changing in any respect, except for the portions that are appended as the document is updated. The encryption of individual blocks

within the document allows different portions of the document to be accessed only by authorized subgroups.

The only change each author needs to make is to update the *b* sequence. In our example, the sequences for the documents created are:

Sequence(Document1): b ={2,...,2, 2,...,2 }
Sequence(Document2): b ={3,...,3, 2,...,2, 2,...,2 }
Sequence(Document3): b ={3,...,3, 3,...,3, 2,...,2, 2,...,2 }
Sequence(Document4): b ={10,...,10, 3,...,3, 3,...,3, 2,...,2, 10,...,10}
Sequence(Document5): b ={1,...,1, 1,...,1, 3,...,3, 3,...,3, 2,...,2, 1,...,1}

Could an author change a sequence entry incorrectly? Certainly. For example, suppose C changes the sequence of *Document5* to be:

Sequence(Document5): b ={1,...,1, 1,...,1, 3,...,3, 1,...,1, 2,...,2, 1,...,1}

However, this does not change the fact that *Text2* was encrypted with *Key3*. It means that *Text2* was first encrypted with *Key3*, and the result was later encrypted with *Key1*. Only the subgroup with *Key3* (who also has *Key1*) can now read *Text2*. The other subgroups will not be able to read the result if they decrypt that portion with *Key1*. Only an author who can decrypt a block can re-encrypt that block with one of his own keys.

Note that it is possible for a recipient of the document to later deny access of a portion of the document to the original author. For example:

Sequence(Document5): b ={1,...,1, 1,...,1, 3,...,3, 14,...,14, 2,...,2, 1,...,1}

Now no one, including the original author, can read *Text2* until either C or D decides to decrypt it with *Key14*. We can assume that the author would archive a copy of his work, to prevent a malicious member of the group from denying him access to it later.

When more subgroups are defined, key management becomes more of a burden. However, the burden rests on the members of the subgroup that share a key. Each subgroup must take responsibility for generating, sharing, storing, updating, and revoking its own keys. In that sense, the key management problem is distributed, rather than centralized.

In addition, we make the assumption that no members of a subgroup will cheat and share a key with a nonmember of the subgroup. This is similar to the common assumption that users will not share their access passwords. Clearly, this assumption is false. Yet, there may be little the subgroup can do to prevent this type of insider attack.

Keeping the Initiator Honest

Suppose we play a game where D thinks of a number between 1 and 10, and A, B, and C try to guess the number. The winner is the one whose guess is closest to the number (without going over).

D secretly wants A to win. D thinks of the number 5, then tells A, B, and C to try to guess the number. A guesses 8, B guesses 4, and C guesses 3. According to the rules of the game, B should be declared the winner. However, D says his number was 9 and declares that A is the winner.

D can cheat another way. D thinks of the number 5, and tells A, B, and C to try to guess the number. He tells A that he is thinking of a number between 1 and 5. He tells B and C that he is thinking of a number between 1 and 100. Clearly, B and C are at a disadvantage.

Now suppose that, before A, B, and C say their guesses, D must publish all the rules of the game and his number (encrypted). The keys belonging to the subgroups are:

Key	Shared by
1	A, B, C, and D
2	D only (at the start of the game)

D composes the following document:

Text1: "I am thinking of a number between 1 and 10. Guess the number. The winner will be the one whose guess is closest to the number (without going over)."
Text2: "My number is 5."

D customizes the authorization for the different portions of the document by encrypting:

Document = Encrypt [Key1, Text1] || Encrypt [Key2, Text2]

That is, he chooses his *b* sequence to be:

Sequence(Document): b ={1,...,1, 2,...,2}

D sends the entire encrypted document to A, B, and C. They can all read *Text1*, but only D can read *Text2*. The players A, B, and C save the document for the duration of the game. After they all make their guesses, D publishes *Key2*. Now they can all read both *Text1* and *Text2*. A, B, and C can verify that D didn't cheat and change his number by applying *Key2* to the original document they received.

In this example, *Key2* has a single purpose and a short lifetime. In this sense, *Key2* is very easy to manage. It is entirely D's responsibility to protect *Key2*. The only reason to use *Key1* is if D wants to choose the players in his game. *Key1* could also be used for a single purpose over a short lifetime.

Although this is just a simple game, we could easily extend this idea to other protocols. For example, we could imagine possible applications such as online silent auctions or perhaps bidding for a contract.

The idea is that it is easy for the author (in this case, D, the initiator of the game) to encrypt different portions of a document with different keys. In this sense, this type of application is similar to the previous application for access control. The primary technical difference between the two applications is the intended lifetimes of the keys.

Expanding a String of Pseudo-Random Numbers

As indicated in the block encryption with multiple-key sets section, there is a possible application for just the intermediate values *Vi* of MK-CTR mode. Recall that these intermediate values are generated by encrypting the current counter value with the key selected by the *b* sequence. This technique is described by Equation 34 and Figure 19.

$$Vi = \text{Encrypt}\,[Ks(bi), \text{counter}+(i-1)] \qquad (34)$$

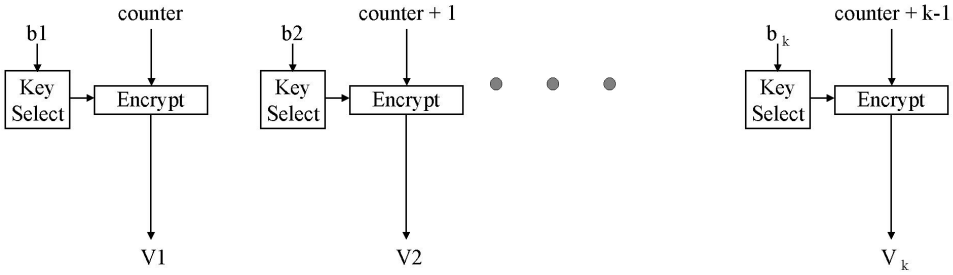

Figure 19: Generation of Intermediate Values Only

We can choose the *b* sequence to be the output of a pseudo-random number generator (PRNG). Each sequence entry *bi* becomes the input to our post-processing expander unit. The output of the expander unit, *Vi*, is a random-looking block of values. The expander unit uses input *bi* to select the key for encrypting the value *ctr+(i-1)*, and outputs *Vi*. This concept is illustrated in Figure 20.

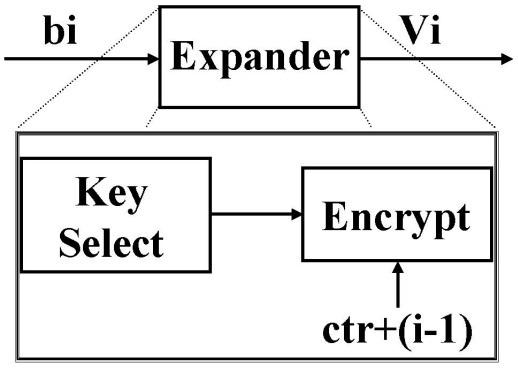

Figure 20: Expanding a Sequence

The expander unit converts a sequence *b* of length *k* into a sequence *V* of length *k*L*, where *L* is the length of the encrypted block.

For example, in AES, *L* = 128 bits. Suppose we continue our previous example with two keys in a set, with key *Ks1* mapped to 0 and key *Ks2* mapped to 1. Hence, for a 10-entry input sequence *b*, the output *V* is 1,280 bits = 160 bytes.

The same expanded sequence can be generated if *b*, the initial counter value, *Ks1*, and *Ks2* are known. However, the expanded sequence *V* would be computationally difficult to re-create without *Ks1* and *Ks2*. It would also be difficult to determine *b* from *V* without either *Ks1* or *Ks2*.

Recall that *V* can be pre-computed before the plaintext message *M* is ready, or before the ciphertext *C* is received. The ciphertext *C* can be generated by $Ci = Mi \oplus Vi$, and the plaintext can be recovered by $Mi^* = Ci \oplus Vi$. Recall that some stream ciphers operate in this manner.

The counter value is updated (typically by adding 1 to the previous value) for each iteration *i*. This means that, even if sequence entry *bi* equals sequence entry *bj*, the output *Vi* does not equal the output *Vj*.

However, if the counter is not updated for each iteration, then duplicate sequence entries will produce the same block of output. For example:

Vi = Encrypt [Ks(bi), counter]
Vj = Encrypt [Ks(bj), counter]

If *bi* = *bj*, then *Vi* = *Vj*. That is, the entire block *Vi* is identical to the entire block *Vj*. In other words, let

$Vi = V_{i1}, V_{i2}, \ldots V_{iL}$
$Vj = V_{j1}, V_{j2}, \ldots V_{jL}$

Then, $V_{i1} = V_{j1}$, $V_{i2} = V_{j2}$, and so on. The entire *L*-bit block repeats. Clearly, in this case the statistical properties of the expanded sequence would not be ideal. Note that ideally a different initial counter value is used for each application of MK-CTR.

Conclusion

This section summarizes the techniques and applications proposed in previous sections, describes the simulation work, includes some concluding remarks, and presents ideas for future work.

Summary

We have proposed and discussed modes of operation that operate on blocks of a plaintext message using different keys for each block. Specifically, MK-ECB mode, MK-CTR mode, and MK-CBC mode were described. The technique of encrypting with multiple keys could be extended to other modes as well. Further, the multiple key encryption technique can be used to create a message authentication code (MAC).

Several applications of these techniques have also been proposed. In particular, we discussed the possibility of using the techniques to convey additional hidden information in each block, using a type of covert channel. In addition to hiding a secret message, the covert channel idea could be used to artificially increase the capacity of the communication channel, or reduce the protocol overhead associated with updating session keys.

We also discussed the application of customizing message encryption at the block level, and demonstrated that this technique could be used to frustrate a certain type of participant cheating in a protocol. Further, we proposed a technique to expanding a sequence of values.

Simulation

The MK-ECB and MK-CTR techniques were simulated with the block cipher AES using MATLAB version 7.0.1.24704 (R14) Service Pack 1, and an AES toolkit [BUCH2001].

The goal of the simulation was to demonstrate the viability of the techniques. A secondary goal was to illustrate the concept via a graphical user interface (GUI). The emphasis of this simulation work was on generating a proof-of-concept, not on creating an efficient implementation.

The GUI makes it easy to see the MK-ECB and MK-CTR techniques in action. The user enters the keys, the key mapping, the sequence entries, an initial counter value (for MK-CTR), and the plaintext. The results are displayed at the click of a button.

Concluding Remarks

In this section, comparisons are made to published works. We also discuss the applicability of these techniques.

Comparisons to Published Works

The author has not found any references to these techniques in the published literature. Therefore, it is possible that this is new work.

The concept is not particularly complex: simply use entries of a sequence to determine which key is used to encrypt or decrypt each block of a message. The sequence, in effect, functions as a different type of "key." Unlike conventional modes of operation, a different key may be used to encrypt each block of the

message. Unlike multiple encryption, a single key is used to encrypt each block. Unlike the technique of interleaving, different keys are used to encrypt each block, and the key to encrypt or decrypt each block is selected by the entries of a sequence. However, one might suggest that the concept is similar to a customizable form of interleaving.

Note that conventional symmetric key block encryption could actually be considered a subset of these multiple key techniques. In conventional block encryption, the key set contains only one key. Therefore, the sequence is always equivalent to $b = 0, 0, \ldots 0$.

Covert channels have been studied for decades. However, the author has found no references to embedding hidden messages in blocks of an encrypted message. To an observer who does not know the secret key, the ciphertext is just random-looking gibberish. It looks like random-looking gibberish whether a hidden message is embedded in it or not. Further, in certain situations, there may be benefit in packing as much information per block as possible.

Other applications exist to customize the encryption of a message. However, these techniques are particularly well suited to customizing the encryption of a message at the block level. In this straightforward application of the modes of operation, the sequence selects the key used to encrypt each block.

A side effect of using multiple keys to encrypt a message is that, even if one key is compromised, the entire plaintext message is not revealed. In addition, if the set contains more than two keys, a single compromised key does not reveal the entire sequence.

Some pseudo-random number generators utilize a strong block cipher such as AES or Triple-DES. However, instead of using a single key or a predetermined pattern of key material to produce the output, the idea presented earlier for expanding a

series of values uses multiple keys. It also allows the keys to be used in a randomly determined pattern, as determined by the sequence.

Applicability

In previous sections, we discussed techniques to reduce the number of times that a block cipher key-scheduling algorithm would need to run. Although these techniques may improve efficiency, this may not be sufficient for certain applications.

For example, consider a data streaming application where data is transmitted or received at speeds of up to 1 to 10 gigabits per second. The relative amount of time needed to schedule keys and "demux/mux" the blocks could be significant.

It is imagined that the mode of operation techniques would be most useful for applications where messages of fixed length are encrypted, then stored locally or transmitted across a communication channel. Intermediate values generated using MK-CTR could be pre-computed and used as an input to a stream cipher. MK-CBC-MAC, like other MACs, could be used to authenticate a message. These example applications are presented in the following table.

Technique	**Applications**
MK-ECB encryption/decryption	Confidentiality of short messages Covert channel for short messages (no re-keying) Customized encryption at the block level
MK-CTR encryption/decryption	Confidentiality Covert channel (including re-keying) Customized encryption at the block level
MK-CTR intermediate value generation	Pseudo-random sequence for stream cipher
MK-CBC encryption/decryption	Confidentiality Covert channel (including re-keying) Customized encryption at the block level
MK-CBC-MAC	Message authentication

These techniques also require multiple keys to be generated, exchanged, stored, and managed. For certain applications, this may be very inefficient. However, for other applications, if at least one key is generated, exchanged, stored, and managed, then it may not be a significant burden to have multiple keys.

If the capability exists to generate one key, then the capability exists to generate multiple keys. Similarly, if one key can be securely shared with a communication partner, then multiple keys can be securely shared. In addition, if each partner already has a mechanism for storing and managing one key, then multiple keys can be stored and managed. If these infrastructure components are already in place, then a significant challenge has already been solved.

It is possible that multiple keys could be exchanged in a single transaction. If the communication partners go to all the trouble of meeting in person, for example, then they might as well hand over multiple keys instead of only a single key. Similarly, in an online exchange, if asymmetric key cryptography is used to exchange a symmetric key, then it may be possible to fit several symmetric keys into a single asymmetric key block. Note that the current size of a RSA modulus is in the range 1024 to 2048 bits. A single 2048-bit RSA encryption could be used to convey up to about 15 128-bit symmetric keys if the keys are concatenated together.

Finally, consider the practicality of storing multiple keys. A 10-round AES key is 128 bits, and a 14-round AES key is 256 bits. Although it is difficult to predict with absolute certainty, it seems likely that secret keys will remain in the 128- to 256-bit range in the near future. Note that one kilobyte of storage can contain 64 128-bit keys.

Future Work

These techniques should be peer-reviewed and analyzed. It would be interesting to investigate the statistical properties of pseudo-random sequences generated using the expansion technique (with MK-CTR) under various conditions. It would also be interesting to explore other applications for these techniques.

Bibilography

[ASN2002] "Abstract Syntax Notation One (ASN.1): Specification of Basic Notation," ITU-T Rec. X.680 (2002) | ISO/IEC 8824-1:2002 (http://asn1.elibel.tm.fr/en/standards/index.htm)

[BLAZ93] M. Blaze, "A Cryptographic File System for UNIX," *1st ACM Conference on Computer and Communications Security,* ACM Press, 1993, pp. 9-16.

[BLAZ94] M. Blaze, "Key Management in an Encrypting File System," *Proceedings of the Summer 94 USENIX Conference*, USENIX Association, 1994, pp. 27-35.

[BOYD88] C. Boyd, "Some Applications of Multiple Key Ciphers," *Advances in Cryptology – EUROCRYPT '88 Proceedings*, Springer-Verlag, 1988, pp. 455-467.

[BUCH2001] Jorg J. Buchholz, "MATLAB Implementation of the Advanced Encryption Standard," http://buchholz.hs-bremen.de/aes/aes.htm, December 19, 2001.

[DESM88] Y. Desmedt, "Subliminal-free authentication and signature," *Advances in Cryptology – Eurocrypt '88 Proceedings*, Springer-Verlag, 1988, pp. 23-33.

[DESM90] Y. Desmedt, "Abuses in cryptography and how to fight them," *Advances in* Cryptology – *CRYPTO '88 Proceedings*, Springer-Verlag, 1990, pp. 375-389.

[ISO89] ISO/IEC 9797, "Data Cryptographic Techniques – Data Integrity Mechanism Using a Cryptographic Check Function Employing a Block Cipher Algorithm," International Organization for Standardization, 1989.

[JANS88] C. J. A. Jansen and D. E. Boekee, "Modes of Blockcipher Algorithms and their Protection against Active Eavesdropping," *Advances in Cryptology – EUROCRYPT '87 Proceedings*, Springer-Verlag, 1988, pp. 281-286.

[KAHN67] D. Kahn, *The Codebreakers: The Story of Secret Writing*, New York: Macmillan Publishing Co., 1967.

[KNUD94] L. R. Knudsen, "Block Ciphers – Analysis, Design, Applications," Ph.D. dissertation, Aarhus University, Nov 1994.

[MERK81] R. C. Merkle and M. Hellman, "On the Security of Multiple Encryption," *Communications of the ACM*, v. 24, n. 7, 1981, pp. 465-467.

[NIST93] National Institute of Standards and Technology, NIST FIPS PUB 46-2, "Data Encryption Standard," U.S. Department of Commerce, December 1993.

[NIST94] National Institute of Standards and Technology, NIST FIPS PUB 186, "Digital Signature Standard," U.S. Department of Commerce, May 1994.

[NIST2000] National Institute of Standards and Technology, NIST FIPS PUB 186-2,"Digital Signature Standard," U.S. Department of Commerce, January 2000.

[NIST2001a] National Institute of Standards and Technology, NIST FIPS PUB 197, "Advanced Encryption Standard," U.S. Department of Commerce, November 2001.

[NIST2001b] National Institute of Standards and Technology, NIST Special Publication 800-38A, "Recommendation for Block Cipher Modes of Operation: Methods and Techniques," U.S. Department of Commerce, December 2001.

[NIST2004] National Institute of Standards and Technology, NIST Special Publication 800-38C, "Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality," U.S. Department of Commerce, May 2004.

[OORS91] P. C. van Oorschot and M. J. Wiener, "A Known-Plaintext Attack on Two-Key Triple Encryption," *Advances in Cryptology – EUROCRYPT '90 Proceedings*, Springer-Verlag, 1991, pp. 318-325.

[RIVE78] R.L. Rivest, A. Shamir, and L.M. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, v. 21, n. 2, February 1978, pp. 120-126.

[RIVE92] R.L. Rivest, "The RC4 Encryption Algorithm," RSA Data Security, Inc., March 1992.

[ROWL2005] C. Rowland, "Covert channels in the TCP/IP protocol suite," *First Monday*, http://www.firstmonday.org/issues/issue2_5/rowland/, 2005.

[SCHN96] B. Schneier, *Applied Cryptography Second Edition: Protocols, Algorithms, and Source Code in C*, New York: John Wiley and Sons, Inc., 1996.

[SIMM84] G. J. Simmons, "The prisoner's problem and the subliminal channel," *Advances in Cryptology: Proceedings of CRYPTO '83*, Plenum Press, 1984, pp.51-67.

[SIMM86] G. J. Simmons, "A secure subliminal channel (?)," *Advances in Cryptology – CRYPTO '85 Proceedings,* Springer-Verlag, 1986, pp. 33-41.

[SIMM94a] G. J. Simmons, "Subliminal channels: past and present," *European Transactions on Telecommunications*, vol. 4, no. 4, Jul. Aug. 1994, pp. 459-473.

[SIMM94b] G. J. Simmons, "Subliminal communication is easy using the DSA," *Advances in Cryptology – EUROCRYPT '93 Proceedings*, Springer-Verlag, 1994, pp. 218-232.