Redvers Consulting Ltd

White Paper

Secure sensitive data with AES encryption - written in COBOL designed for COBOL

Contents

Executive Summary

This paper isn't going to try to sell you the concept of data encryption. If you're reading this, you already know you need to encrypt confidential information. What this document will do, is help you fit the encryption process simply and securely into COBOL applications with minimal disruption to existing systems.

Essentially, encryption protects data that needs to be kept secret by combining it with an apparently random string of bits. A perfectly valid random string could be derived from a room full of people, repeatedly flipping a coin, recording the results. Fortunately, labour saving devices known as computers can be used to produce a pseudo-random string with greater reliability and at much less expense.

It's the secure, reliable production of these pseudo-random strings that the *Advanced Encryption Standard* (AES) was designed to address.

What is AES encryption?

AES encryption is based on the Rijndael algorithm, developed by two Belgian cryptographers: Vincent Rijmen and Joan Daemen.

In November 2001 the U.S. National Institute of Standards and Technology[1] (NIST) selected the Rijndael algorithm as the approved Advanced Encryption Standard (AES) algorithm (FIPS PUB 197[2]) for federal government encryption.

Later, in June 2003 the National Security Agency[3] (NSA) declared AES encryption as "all key lengths of the AES algorithm (i.e., 128, 192 and 256) are sufficient to protect classified information up to the SECRET level" and "TOP SECRET information will require use of either the 192 or 256 key lengths.".

AES is also highly suited to processing credit and debit card payments under the Payment Card Industry Data Security Standard[5] (PCI DSS).

Why use COBOL?

As the majority of financial services systems are written in COBOL, this document will focus on the installation of AES encryption within COBOL applications.

Note: Only an elementary knowledge of encryption and COBOL are necessary to understand the points made in this paper.

Alternatives to COBOL AES Software

Hardware or software?

Hardware encryption on peripheral devices:

- ✗ Additional hardware complexity and maintenance costs.
- ✗ A hardware failure could leave you with your data encrypted with no means to decrypt it.
- ✗ All the data written to the device will be encrypted, not just the sensitive items.
- ✗ All data is decrypted when read, even for the most trivial or insecure inquiry.
- ✗ Full volume encryption and decryption will increase the total processing requirement and therefore increase electricity and cooling costs.
- ✓ No application program changes.

Software encryption initiated by applications:

- ✗ Minor application code changes required.
- ✓ Only the information that needs to be kept secret is encrypted.
- ✓ The majority of application data remains in place and accessible as before.
- ✓ Multiple encryption modes, keys and key lengths can be used.
- ✓ Easy to upgrade the encryption algorithm.
- ✓ Changes to encryption keys can be implemented incrementally.

Which language is best for encryption?

- ✗ **Assembler** routines can offer good efficiency returns for small scale operations, however the AES algorithm is a long and complex series of mathematical instructions, not suited to assembler. An optimised, COBOL program, compiled using a good quality compiler will almost certainly be faster.
- ✗ **Java** requires approximately 10 times the computing power for the same logic written in COBOL. Even if this load is passed to a speciality engine, total CPU and elapsed times will still be an issue.
- ✓ **COBOL** subroutines, called from a COBOL application, will produce the most efficient and compatible solution, requiring minimal CPU and no additional staff training.

Off-the-Shelf or Bespoke?

As the AES algorithm is a global standard sequence of complex data manipulation instructions using binary level mathematics, it makes sense to use commercially available encryption/decryption subroutines. A quality software vendor will also have optimised the code and support any future enhancements to the AES algorithm, should they become necessary.

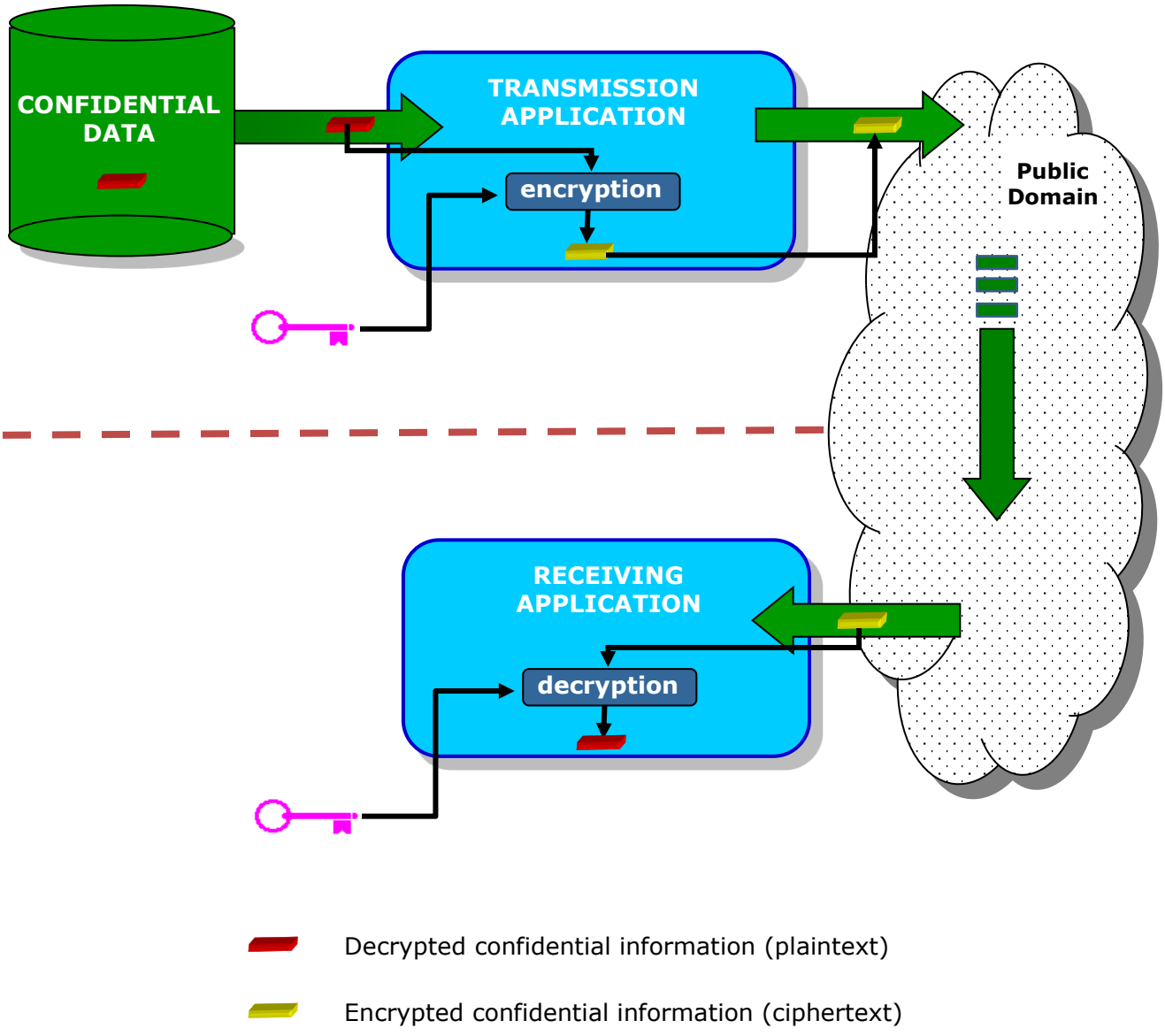
COBOL Encryption for Transfer

COBOL AES encryption ensures the secure transfer of information by replacing confidential data with encrypted strings within the transmission file. On receipt of the file, trusted parties use AES decryption to recover the readable plaintext.

Applications creating a transmission file, pass confidential fields (**plaintext**) with the encryption key to the encryption subroutine, which returns the corresponding encrypted string (**ciphertext**).

Applications receiving a transmission file, pass encrypted strings (**ciphertext**) with the same encryption key to the decryption subroutine, which returns the confidential data (**plaintext**).

As AES is an industry wide standard, non-COBOL applications will be able to decrypt ciphertext created by COBOL applications and vice-versa, if the same key and confidentiality mode is used. However, differences in character encoding (EBCDIC/ASCII) must be taken into account.



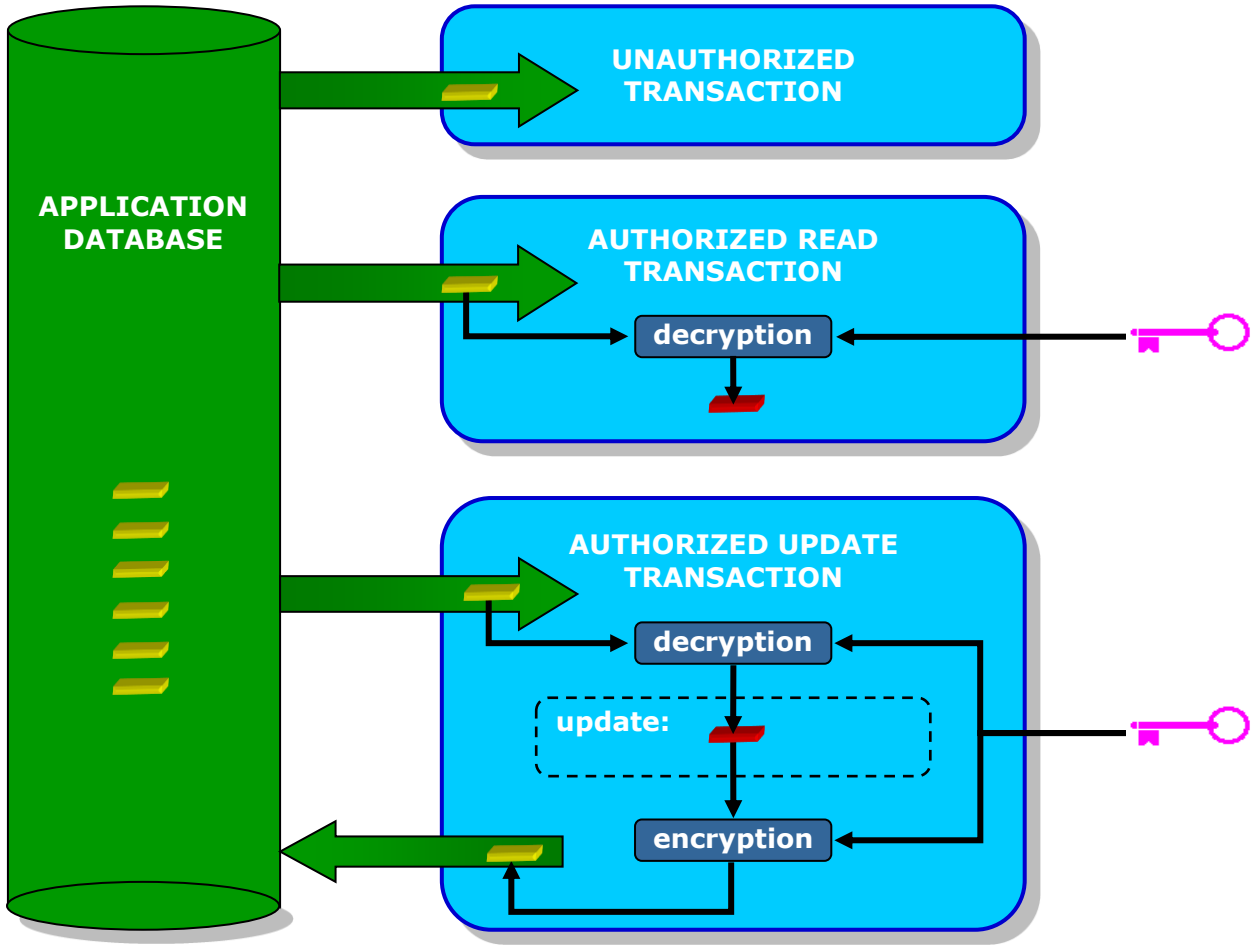
COBOL Encryption at Rest

COBOL AES encryption at rest will require a one-off procedure to replace all fields containing confidential information (**plaintext**) with their equivalent encrypted strings (**ciphertext**). After this process, application databases and files will remain intact and operational as before. For the majority of applications, the decision to encrypt confidential data will require no program changes.

Unsecured transactions will continue to access records containing encrypted fields in the usual way; they just won't be able to interpret the ciphertext field content.

When an application has authorization to access encrypted data, it passes the ciphertext and encryption key to a decryption subroutine which returns the readable plaintext.

If an authorized application needs to update encrypted data, it follows the usual decryption method and updates the plaintext. The updated plaintext is then passed to the encryption routine (with the encryption key) and the new ciphertext output is written back to the database or file.



Encrypted confidential information (ciphertext)

Decrypted confidential information (plaintext)

AES Encryption Strategy

AES Properties

- Block cipher – working on blocks of 128 bits (16 characters) at a time.
- Symmetric-key algorithm - the same key is used to encrypt and decrypt.
- Encryption key can be 128, 192 or 256 bits in length.
- Algorithm is a publicly known, logical procedure – only the encryption key needs to be kept secret.

The final point in the above list is of particular relevance to COBOL applications management. It means that although the key needs to be kept secure, the program source code can be kept in standard, accessible libraries – a common scenario for COBOL application programs.

AES encryption isn't just a one-size-fits-all solution. It has several modes of operation and other features to help it fit seamlessly into a wide range of computer applications. The choice of mode, key length and other parameters will depend on the application involved, so the following points need to be considered before embarking on an AES project.

Points to Consider

You may wish to use the following questions as a way to select only the sections of this document most relevant to your project:

Should my ciphertext be the same or different for a given plaintext?

When encrypting unique key fields, the same ciphertext for a given plaintext will be required to maintain the integrity of the application. For example, a credit card number would need to produce a consistent ciphertext if card transactions for the card are to be linked to the correct card owner details - see **Consistent Ciphertexts**.

For non-key data, applications will be more secure with a different ciphertext for a given plaintext. For example, if expiry dates within a credit card application were to be encrypted using consistent ciphertexts, knowledge of the true expiry date of one credit card would reveal the expiry date of all cards expiring on the same date - see **Inconsistent Ciphertexts**.

What if I don't need to decrypt the ciphertext after encryption?

Verification of user id's, passwords or PIN numbers are the most common conditions where decryption isn't required. Indeed, it may even be possible to use a standard hashing routine instead of encryption. If the input, after encryption or hashing, matches the previously encrypted/hashed id, password or PIN on file, then the input is verified. It's not necessary for the application to derive the actual id, password or PIN in use - see **Encryption without Decryption**.

Which key length should I use?

The choice of 128, 192 or 256 bit key lengths will depend on the security level required for the data involved. 128 bits is usually regarded as sufficient for most commercial applications.

Processor requirements for a 192 bit key will be approximately 20% greater than a 128 bit key and a 256 bit key will require about 40% more CPU than a 128 bit key.

Can I generate ciphertext for decryption by an external AES encryption routine (or visa-versa)?

Yes. AES encryption is a global industry standard regardless of platform or programming language. However, the encryption key, confidentiality mode (see **Appendix A: Confidentiality Modes**) and, if required, Initialization Vector (see **Appendix B: Initialization Vectors**) all need to be the same for encryption and decryption, so these will need to be agreed with the external party beforehand.

Can I transmit ciphertext in an XML document?

No - not even within a CDATA section. Ciphertext consists of seemingly random bit patterns that could, by chance, represent any character or string of characters. For example, the "<" character could be created once in every 256 characters of ciphertext ($1:2^8$) and the characters "]]>" are likely to appear once in every 17MB of ciphertext ($1:2^{24}$). Also, XML has a restricted character set[8], so invalid characters appearing in ciphertext will mean the XML isn't "well-formed".

The most common solution is to convert the ciphertext to Base64[9] format (characters in the range A–Z, a–z, 0–9, "+", "/" and "="). This can either be done within the AES encryption subroutine or by using a separate software tool to convert ciphertext to Base64. E.g.: RCBINB64[10].

Conversion to Base64 will cause an unavoidable expansion in the ciphertext length of about 33%.

An alternative to Base64 would be to produce a format-preserved ciphertext using an alphanumeric alphabet, see **Format Preserved Ciphertexts**.

Can I compress ciphertext?

You could try but the random bit patterns in ciphertext will probably produce a compressed length, longer than the uncompressed length. However, you can encrypt compressed plaintext. Therefore, the sequence of events should be:

compress, *encrypt*, → *decrypt*, *decompress*.

What if I need the ciphertext in a specific format and length?

Ciphertext can usually be stored in alphanumeric COBOL fields (`PIC X`) without any problems. However, if a field to be loaded with ciphertext is defined as numeric or if it must conform to a specific length or format, then there will be formatting issues (ciphertext is a random array of bits). For the storage of ciphertexts in fields with restricted formats or values, see **Format Preserved Ciphertexts**.

How can I encrypt and decrypt large data volumes very quickly?

Applications that need to process high transaction volumes, very fast, may encounter performance problems when running the AES algorithm at peak processing times. To avoid this problem, a secure reserve of pseudo-random binary strings can be built during off-peak hours. This reserve can then be used to encrypt and decrypt

confidential data just by using *Exclusive Or* (XOR) logic (**Appendix E: Exclusive Or**) at busy times - see **Binary Banks** section for more details.

Is there anything else I need to know?

The safe governance of encryption keys should be regarded as a top priority for applications management. As the circumstances within every installation will be different and the details of how keys are managed must be kept secret, a publicly available white paper isn't the best place to detail how encryption keys should be stored or handled.

Having said that, encryption keys are more likely to survive a Brute-force attack[11] if they consist of an unpredictable bit pattern rather than printable characters. One way to do this might be to use the output from a hash process (SHA-2 or SHA-3) or to start with an alphanumeric string and then pass this through AES encryption, using the output as the actual encryption key.

As part of good practice, we also suggest initializing any application fields containing encryption keys or plaintext immediately after the encryption/decryption activity has completed. Good quality encryption software should have a facility to initialize any of its storage areas containing key or plaintext information.

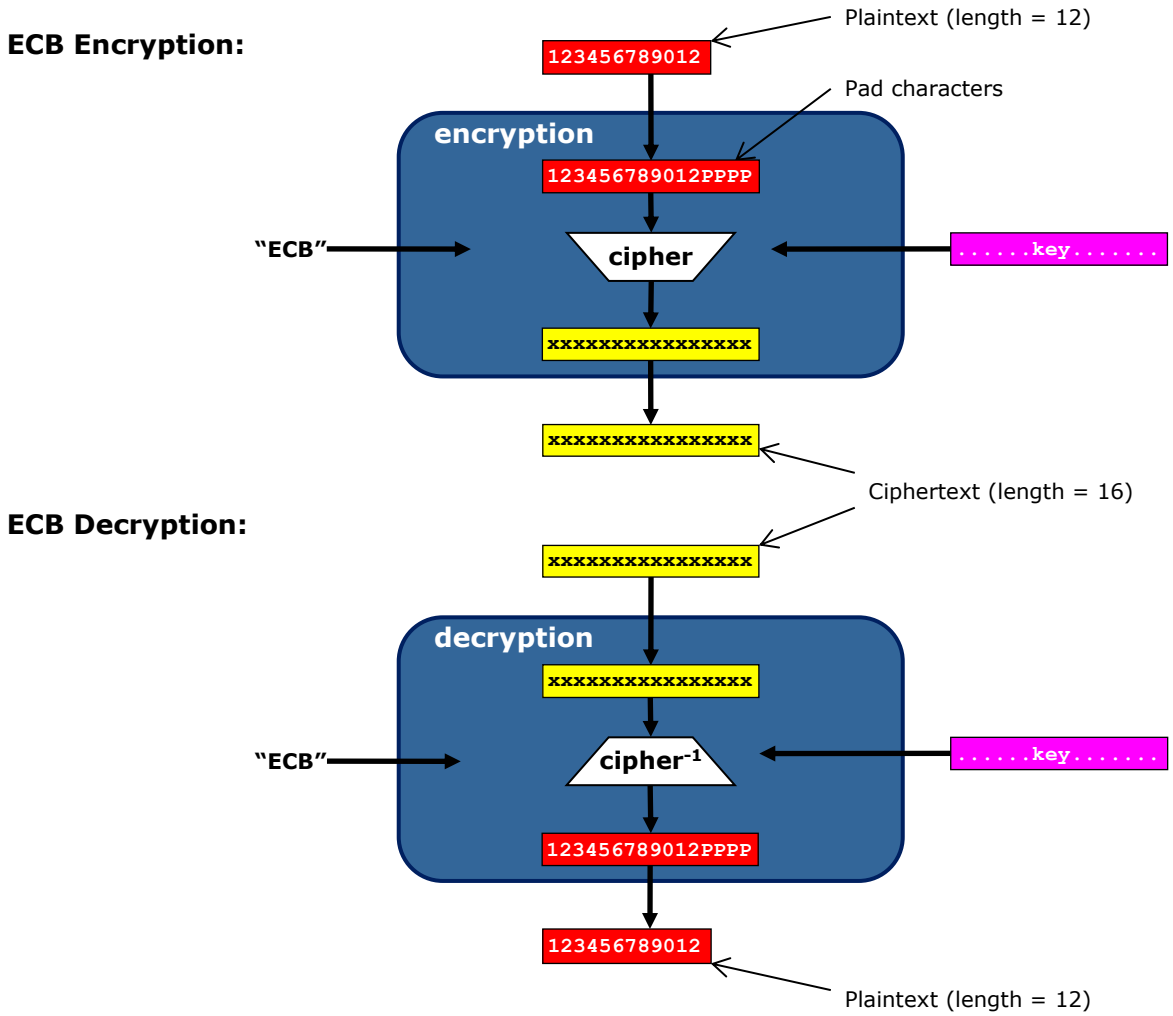
In the event an encryption key is compromised, a new key will need to be created and all encrypted fields re-encrypted as soon as possible. In non-urgent circumstances it may be possible to adopt a phased migration to the new key, using the date-time of the last encryption to identify whether the old or new key is to be used for decryption, then, if necessary, re-encrypt using the new key.

Consistent Ciphertexts

If AES encryption is to produce a consistent ciphertext string, for a given plaintext and key, the AES encryption cipher must be used in conjunction with the Electronic Code Book (ECB) mode of operation (see **Appendix A: Confidentiality Modes**).

Using ECB Mode

To encrypt a plaintext, the application passes the mode ("ECB"), encryption key and plaintext to the encryption subroutine. The subroutine then returns the equivalent, consistent, ciphertext. To decrypt, the same mode and key are passed to the decryption routine with the ciphertext and the plaintext is returned. See below:



The main disadvantage in using ECB mode, is that it never produces a ciphertext the same length as the plaintext. At least one pad character (see **Appendix D: Pad Characters**) must always be added to the right of the plaintext so that the decryption routine can determine the length of the original plaintext. In addition, plaintext must be passed to the AES cipher in complete blocks of 16 bytes. The combination of these two conditions will always result in a ciphertext length expanded to the next multiple of 16 (even if the plaintext was already a multiple of 16).

If decryption is required, every ciphertext character will be needed for input to the decryption process, so the expansion of ciphertexts must be taken into consideration.

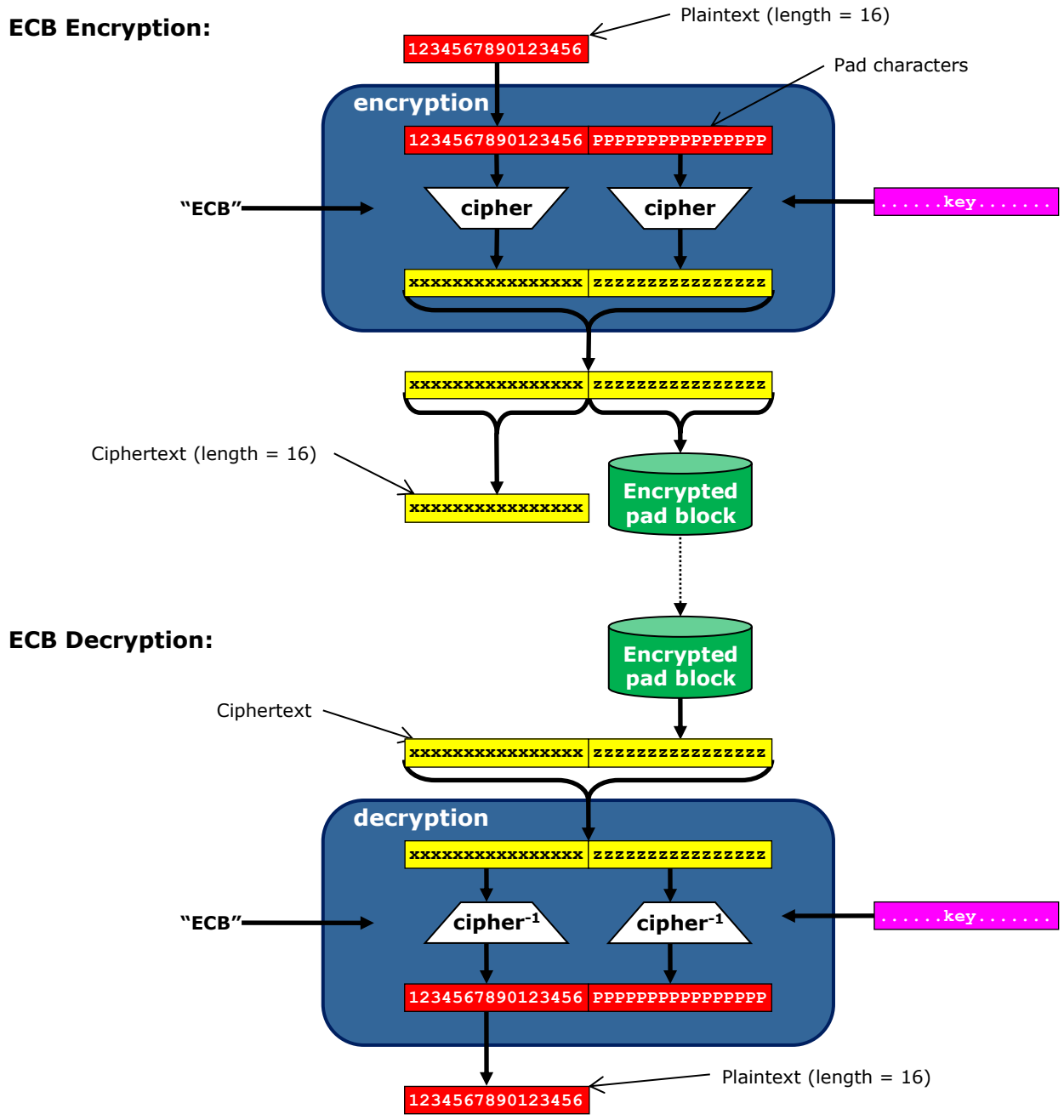
If it's easy to store the complete ciphertext string within the application, then this is all that's needed to provide the necessary input for successful decryption. However, if disk space is at a premium or if file layouts can't be changed, the techniques outlined in the following pages may help to reduce the effects of ECB ciphertext expansion.

Removing the Pad Character Block

If a plaintext always has the same length, and this length happens to be an exact multiple of 16, the application can avoid ciphertext expansion by storing a single copy of the encrypted pad characters that make up the final block of ciphertext (see **Appendix D: Pad Characters**).

As AES encryption operates on complete blocks of 16 bytes at a time, working from left to right, this final pad block has no influence on the ciphertext to the left, and it will always be the same for a given key. Therefore, only one copy of the encrypted pad block need be stored for each encryption key. All subsequent encryptions on plaintext lengths known to be a multiple of 16, for that key, can discard the rightmost 16 characters of ciphertext, leaving the remaining ciphertext length the same as the input plaintext length.

When a ciphertext without its rightmost pad block needs to be decrypted, the single stored copy of the encrypted pad block must be appended to the right of the ciphertext. Decryption, using the reverse cipher, can then be completed as normal. See below:



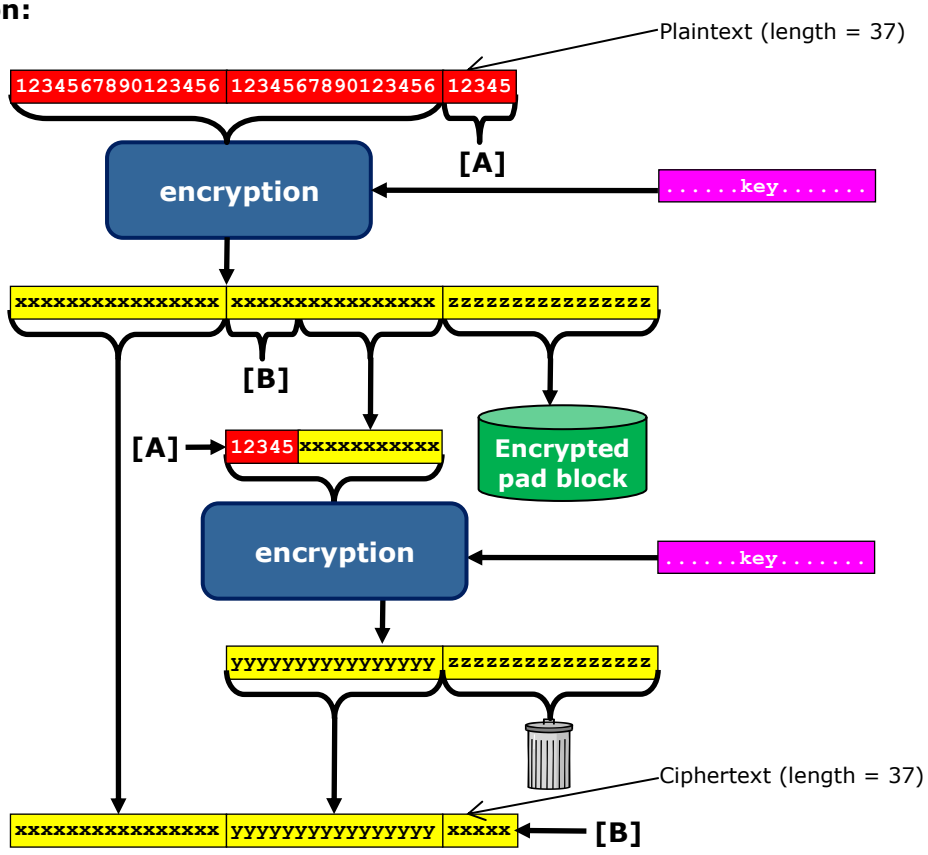
Ciphertext Stealing

For ECB encryption of plaintext lengths not exact multiples of 16 but greater than 16, ciphertext expansion can be avoided by using a technique known as ciphertext stealing[12].

The encryption ciphertext stealing procedure is as follows (see diagram below):

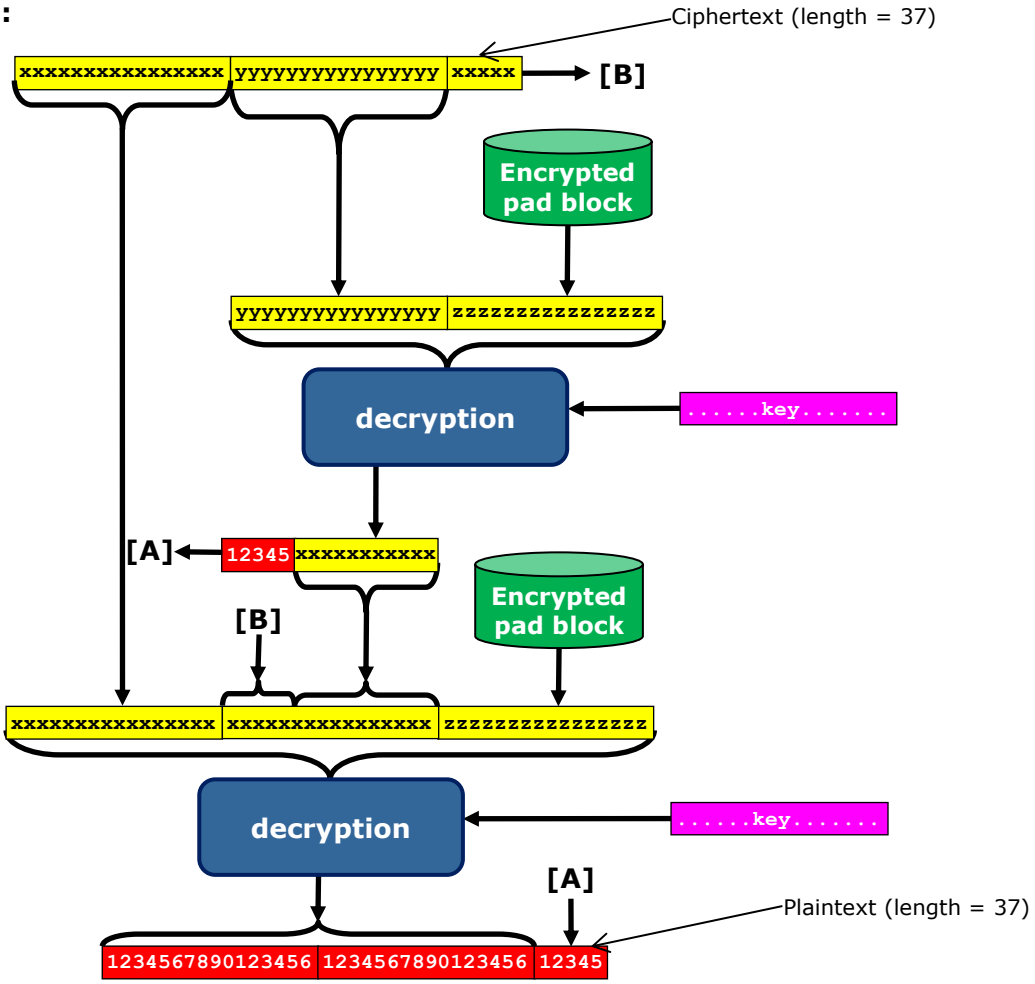
1. Move the rightmost bytes of plaintext that are not part of a complete block of 16 characters to program storage area **[A]**.
2. ECB encrypt all the plaintext except the characters moved to storage area **[A]**. This encryption will produce an additional 16 byte encrypted pad block - see previous **Removing the Pad Character Block** section.
3. Determine the number of characters placed in storage area **[A]** and move this number of bytes from the start of the last 16 byte ciphertext block, to a second storage area **[B]**.
4. Overwrite the ciphertext bytes, saved in **[B]** (step 3) with the plaintext bytes, saved in **[A]** (step 1).
5. ECB encrypt the last 16 byte block of ciphertext (this block will include some bytes "stolen" from the previous encryption). Again, an additional encrypted pad block will be produced. This block can be discarded because it will be the same as the encrypted pad block stored in step 2.
6. Overwrite the original last ciphertext block with the ciphertext block produced in step 5.
7. Finally, add the characters in storage area **[B]** to the end of the ciphertext.

ECB Encryption:



The decryption ciphertext stealing procedure is a reversal of the encryption procedure - see diagram below:

ECB Decryption:



Note: This process will produce a ciphertext with some characters double encrypted and several character positions out of position. None of this matters as long as the ciphertext stealing procedure is run in reverse when decrypting.

Other Plaintext Lengths

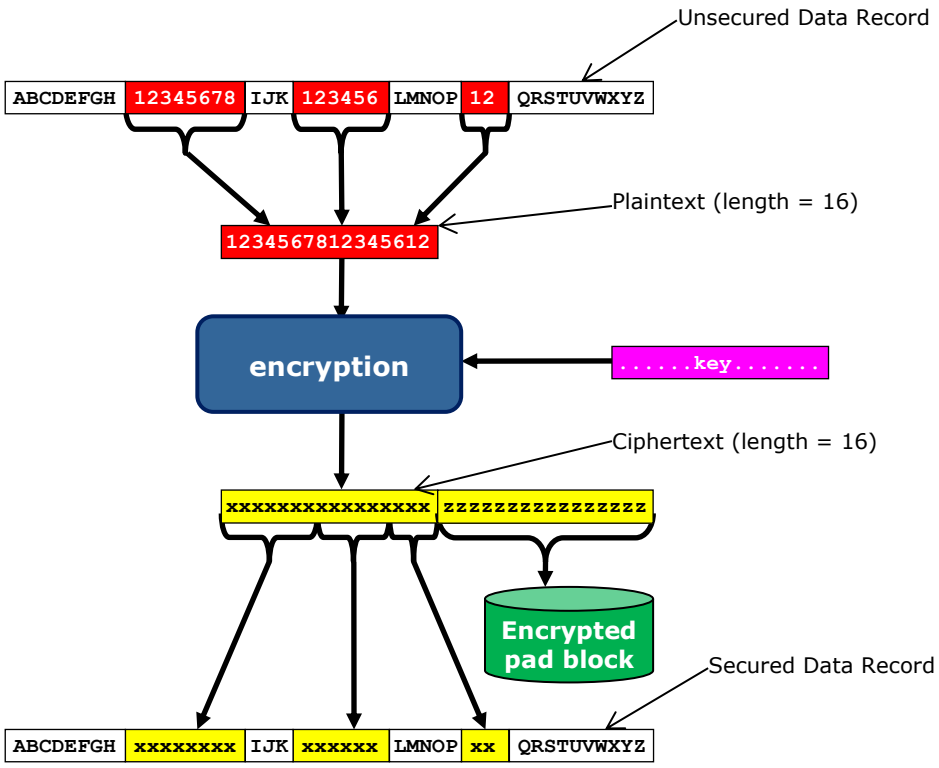
For ECB encryption and decryption of plaintext lengths 1 to 15 bytes, the involvement of some additional application fields will be required if ciphertext expansion is to be avoided. This is because a 16 byte block is the minimum input to the AES encryption/decryption cipher.

Ciphertext blocks don't need to be stored as one contiguous field within the application. They can be broken up and stored in separate smaller fields, then recombined when required for decryption.

Therefore, the simplest solution to short plaintexts is to encrypt a little more information than the absolute minimum. For example, if an eight byte account number is to be encrypted, why not also encrypt the bank sort code and/or account type at the same time. Once the combined length of a group of fields is more than 15, a complete plaintext block can be built by the application and used as input to the encryption routine.

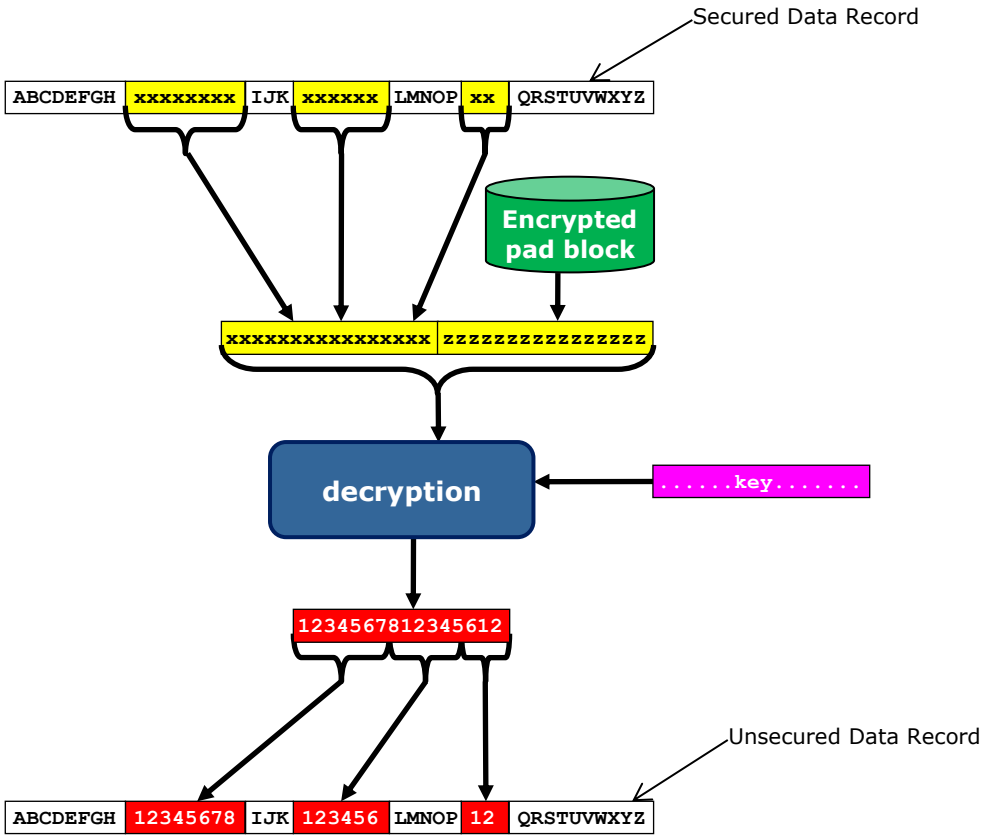
The **Removing the Pad Character Block** and/or **Ciphertext Stealing** techniques can also be used to produce a ciphertext the same length as the plaintext. The ciphertext can then be broken up across the field group and stored in place of the original plaintexts. See below:

ECB Encryption:



For decryption, the application must rebuild the complete ciphertext from the contents of the field group and pass this block to the decryption routine (with the encrypted pad block, if it was removed during encryption). Plaintext for all fields within the group will then be available. See below:

ECB Decryption:



A variation on selecting a group of different fields within a given record/row, would be to select the same field from several different but related records/rows. Under this scenario, multiple occurrences of the field would be encrypted and decrypted simultaneously for all records/rows in the related group.

Note: Do not be tempted to try to run ECB encryption with partial plaintext/ciphertext blocks or to produce a common ciphertext and then use *exclusive or's* to produce shorter ciphertexts for specific fields. This will either not work or produce ciphertext highly vulnerable to unauthorized decryption.

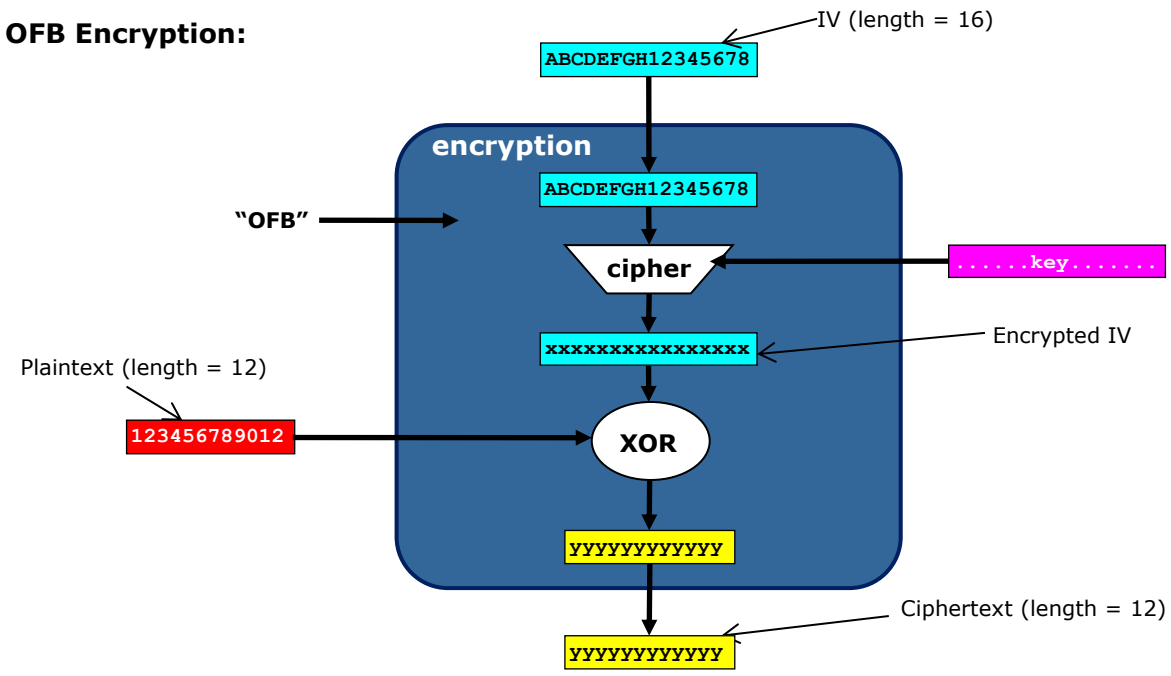
Inconsistent Ciphertexts

The production of an inconsistent ciphertext from the same plaintext and key, relies on the AES encryption cipher being used in conjunction with a mode of operation other than Electronic Code Book (ECB) mode (see **Appendix A: Confidentiality Modes**). These confidentiality modes require an Initialization Vector (IV) (see **Appendix B: Initialization Vectors**) or Counter block (see **Appendix C: Counters**) to introduce an element of unpredictability to the encryption process.

Using OFB Mode

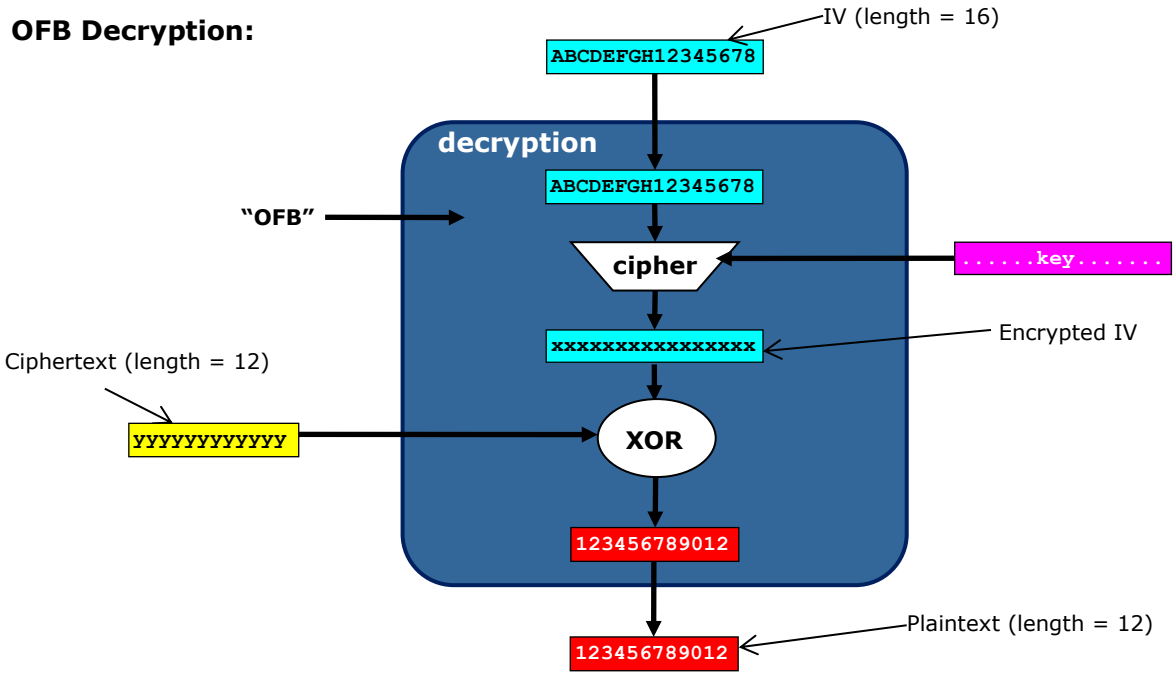
Of the available inconsistent confidentiality modes, Output FeedBack (OFB) is frequently regarded as the most appropriate for COBOL applications. This mode will produce a ciphertext the same length as the plaintext (making it easy to replace plaintext fields with the equivalent ciphertext) and the IV doesn't need to be unpredictable – just unique. In addition, the more efficient forward cipher is used within both the encryption and decryption subroutines.

To encrypt a plaintext in OFB mode, the application passes the IV, mode ("OFB"), encryption key and plaintext to the encryption subroutine. The subroutine then returns the equivalent, inconsistent, ciphertext – see example below:



The diagram above highlights the importance of the Initialization vector (IV). It is the IV that is actually passed through the AES cipher, not the plaintext. To produce the ciphertext, each plaintext character is combined with an encrypted IV character, using *exclusive or* (XOR) logic (see **Appendix E: Exclusive Or**). Any unused characters in the encrypted IV are discarded.

For decryption, the same IV, mode and encryption key are passed to the decryption routine with the ciphertext. The plaintext is then returned – see below:



As shown above, the decryption routine recreates the encrypted IV and then combines this with the ciphertext using the *exclusive or* (**XOR**) logic which recovers the plaintext.

The combination of using optimised COBOL subroutines with the AES forward cipher, will result in efficient encryption and decryption. However, as the AES cipher works on blocks of 16 bytes at a time (regardless of the plaintext length) it may be possible to derive even greater performance by combining multiple small fields into a single plaintext block - see examples in the previous **Other Plaintext Lengths** section (ignoring pad characters). A single execution of the cipher can then be used to encrypt or decrypt several small fields simultaneously.

Note: An additional consideration when building inconsistent ciphertexts is that they introduce the possibility of ciphertext "*collisions*". Ciphertext "*collisions*" are when two different plaintexts, produce the same ciphertext when encrypted. This will not be a problem for most applications but system designers should be aware of this possibility, in the event that ciphertext is considered to be used as a key field.

Encryption without Decryption

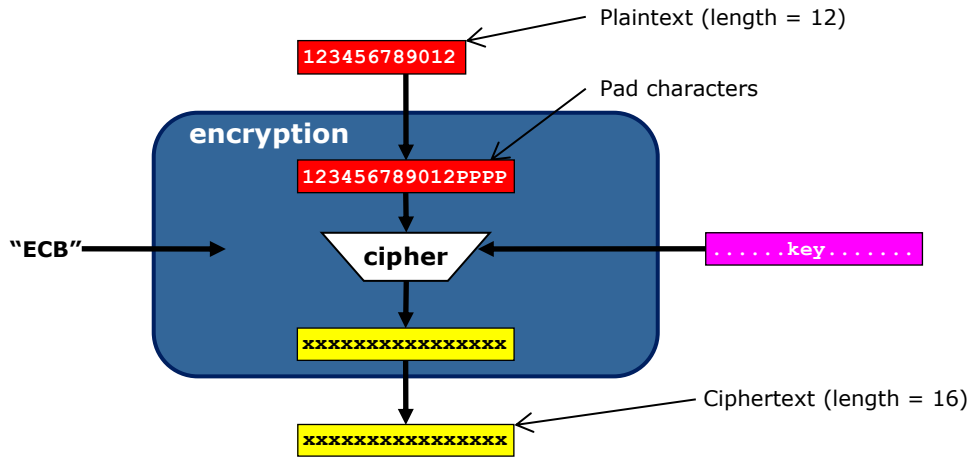
When input such as a user id, password or PIN number needs to be authenticated, it isn't usually necessary to decrypt the information held in computer storage. A one-way encryption or a hash total of the input is all that's needed for comparison with the information held on file, resulting in a "verified" or "not-verified" outcome.

One-way Encryption

When authenticating with one-way encryption, it's vital that the same ciphertext is generated for a given plaintext. Therefore, the AES encryption cipher must be used in conjunction with the Electronic Code Book (ECB) mode of operation (see **Appendix A: Confidentiality Modes**).

To encrypt a plaintext, the application passes the mode ("ECB"), encryption key and plaintext to the encryption subroutine. The subroutine then returns the equivalent consistent ciphertext - shown below:

ECB Encryption:



An aspect of ECB encryption is that pad characters (see **Appendix D: Pad Characters**) are always added to the right of the plaintext so complete blocks of 16 bytes are passed to the encryption cipher. This produces a ciphertext length expanded to the next multiple of 16 (even if the plaintext was already a multiple of 16).

This ciphertext expansion can be a problem if the ciphertext is to be stored in place of the original plaintext. However, the complete ciphertext is only necessary as an input to decryption. If the application simply needs to verify, beyond reasonable doubt, that the correct user id, password or PIN number was entered, a comparison with a selection of ciphertext bytes is all that's necessary.

For example, a 4 digit PIN, passed through ECB encryption would produce a ciphertext of 16 bytes. If only ciphertext byte positions 6 through 9 are stored in the application, the resulting bit pattern is likely to be unique for each of the 10,000 different PIN possibilities.

Note: Input verification using confidentiality modes other than ECB is also possible but the same Initialization Vector (IV) will be required to produce the consistent ciphertext. This is the only scenario where IV's should ever be reused. If the ciphertext created doesn't match with the ciphertext on file, the failed ciphertext must be destroyed. When a new password or PIN is to be encrypted, a new, unique IV must be used (see **Appendix B: Initialization Vectors**).

Hash Total

An alternative to using one-way AES encryption for input authentication is to pass the plaintext id, password or PIN through one of the standard hash totalling algorithms: SHA-1, SHA-2 or SHA-3. These algorithms don't require a mode, key or initialization vector, they just produce a consistent binary hash total, sometimes called a "message digest", dependant on the bit values and their positions within the plaintext string.

Hash totals appear similar to ciphertext but they cannot be used to recover the original plaintext. This is because the hashing algorithm destroys information required to reverse the hashing process.

The length of the hash total will depend on the algorithm used. With the exception of SHA-1, which produces a hash total 160 bits (20 bytes) long, the number in the extended SHA name refers to the number of bits in the hash total result. Therefore, SHA-224 produces a hash total 224 bits (28 bytes) long and SHA-512 produces a hash 512 bits (64 bytes) long. The longer the hash, the less chance of a hash total "collision" i.e.: when two different message texts produce the same hash total.

As described in the **One-way Encryption** section, if the length of the hash total is too long to be easily stored in the application, some of the right-most character positions can be removed. The application only needs to verify, beyond reasonable doubt, that the correct user id, password or PIN number was entered, therefore a binary hash total truncated to the length of the id/password/pin field should be sufficient.

Note: SHA-1, SHA-2 and SHA-3 hash totals can be generated by the Redvers Hashing Algorithm[13].

Format Preserved Ciphertexts

There are several methods of producing ciphertext in a specific format and each method has the usual trade-offs between demands on CPU, storage and uniqueness of result. Methods suitable for COBOL applications tend to fall into one of three groups: **Ciphertext Sort**, **Radix Division** and **Feistel Networks**.

Ciphertext Sort

These methods consist of building a simple look-up table for every possible plaintext-ciphertext pair and sorting the table into ciphertext sequence.

Advantages:

- ✓ Removes the risk of ciphertext "*collisions*" in the formatted ciphertext (when two different unformatted ciphertexts, produce the same ciphertext when formatted).
- ✓ The tasks of encryption and decryption are moved to a one-off batch event, rather than being invoked repeatedly during the lifetime of the encryption key.

Disadvantages:

- ✗ ECB encryption mode must be used to produce a consistent one-to-one relationship between each plaintext and its ciphertext. This may have security implications – see **Points to Consider**.
- ✗ The one-off batch process may require high computer resources if the range of possible formatted outcomes is large.
- ✗ The final cross-reference table must be stored securely.

Method:

This method requires a one-off batch process as follows:

1. Build a table or file [**X**] consisting of every possible valid preserved format plaintext (usually in ascending sequence but could be in any sequence).
2. For each entry in table/file [**X**], pass the plaintext through AES encryption running ECB mode using the same key and store the resulting ciphertext with its corresponding plaintext in a new table/file [**Y**].
3. Sort table/file [**Y**], into ciphertext sequence.
4. Simultaneously read through each entry in table/file [**X**] and table/file [**Y**], starting from entry 1, building a third table/file [**Z**], consisting of the plaintext from table/file [**X**] and the plaintext from table/file [**Y**].
5. Discard table/files [**X**] and [**Y**].

The application now has a cross-reference table for the encryption or decryption of any plaintext or formatted ciphertext, by looking up the corresponding entry in table/file [**Z**].

Note: The cross-reference table [Z], created above, must be kept fully secure from outside access.

Example 1:

The requirement is to encrypt and decrypt US state codes to codes that conform to the range of valid US state codes. The three tables in the method would look like the below (ciphertext shown in hex):

Table:X
AK
AL
AR
….
WI
WV
WY

Table: Y (before sort)	
AK	9ac644070198fc55 78f2b357323cbf62
AL	ff98c0d05e3dad3e b3d6236f23e74196
AR	47d2e1bf72264fa0 1fb274465e56ba20
….	….
WI	3a361407219844ad 48f2750932ec81a3
WV	067ae50986c93774 8176b1a8a482650d
WY	b1a8a4820d052e1b 9a2614f23061320d

Table: Y (after sort)	
WV	067ae50986c93774 8176b1a8a482650d
WI	3a361407219844ad 48f2750932ec81a3
AR	47d2e1bf72264fa0 1fb274465e56ba20
….	….
AK	9ac644070198fc55 78f2b357323cbf62
WY	b1a8a4820d052e1b 9a2614f23061320d
AL	ff98c0d05e3dad3e b3d6236f23e74196

Table: Z	
AK	WV
AL	WI
AR	AR
….	….
WI	AK
WV	WY
WY	AL

Note: There is a reasonable chance that one or more state codes may appear in the same position within table X as the sorted table Y, resulting in no change from encryption or decryption (see code "AR" above). This doesn't matter, as long as no adversary knows which codes have changed and which have not.

Example 2:

The requirement (a PCI requirement) is to encrypt and decrypt positions 7 through 12 of credit card numbers to 6 encrypted numeric characters, with no collisions, so that the encrypted numbers can be stored in place of the original credit card numbers. The three tables would look like the below (ciphertext shown in hex):

Table: X
000000
000001
000002
….
999997
999998
999999

Table: Y (before sort)	
000000	8d052e1b9a2614f2 b1a8a4823061320d
000001	0671a8a482ae5098 6c937748176b650d
000002	fad3eb3d623f98c0 d05e3d6f23e74196
….	….
999997	46ba27d2e1bf7226 01fb2744fa465e50
999998	3a340724ad48f275 0932e611984c81a3
999999	bf69a198fc5578f2 b35c6440707323c2

Table: Y (after sort)	
000001	0671a8a482ae5098 6c937748176b650d
999998	3a340724ad48f275 0932e611984c81a3
999997	46ba27d2e1bf7226 01fb2744fa465e50
….	….
000000	8d052e1b9a2614f2 b1a8a4823061320d
999999	bf69a198fc5578f2 b35c6440707323c2
000002	fad3eb3d623f98c0 d05e3d6f23e74196

Table: Z
000001
999998
999997
….
000000
999999
000002

Note: In the previous example, storage requirements have been kept to a minimum by only storing the table Y value in table Z. This can be done in the case of numeric plaintexts if table X is built in plaintext sequence. The value from table X is merely its position number, – 1. Encryption can be achieved using the 6 digit plaintext, + 1, as a subscript to directly look-up the ciphertext value. Decryption must search for a matching table Z value and use the resulting subscript, - 1, as the plaintext. If decryption efficiency is important, a reverse table Z can be created by sorting table X values and table Y values into table Y sequence – then discarding the table Y values.

Radix Division

This method uses COBOL's ability to interpret ciphertext bit strings as integers through the use of binary (COMP) field definitions. The general concept is to divide the integers representing the ciphertext by the number of possible valid outcomes for each character position (known as the *radix*), using the remainder to look up a valid character in a table for the position.

Advantages:

- ✓ Flexible: each formatted character position is built independently, allowing for a mix of different character ranges depending on its position within the field.
- ✓ Minimal additional processing.

Disadvantages:

- ✗ The original AES ciphertext output must also be stored within the application to enable decryption (ciphertext need not be stored in secret).
- ✗ Risk of ciphertext "*collisions*" (when two different unformatted ciphertexts produce the same ciphertext when formatted) – the importance of this will depend on whether the encrypted field is to be used as a key or not.

Method:

This method consists of the following initial step:

1. Break up the ciphertext into 8, 4, 2 or 1 byte strings and move these strings, right-justified, into fields pre-filled with low-values and redefined as numeric unsigned binary. *Exactly how this is done, is going to depend on the length of the ciphertext and the compile options in use. The desired outcome is to have a series of roughly equal length binary fields containing the decimal equivalent of the ciphertext bit settings.*

Then, for each formatted character position:

2. Determine the radix for the formatted character position (may not be the same for all character positions). E.g.: Upper case alphabetics would have a radix 26.
3. Build a table consisting of each valid output character. E.g.: "A", "B", …. "Z".
4. Divide the left-most binary field by the radix, replacing the original binary field value with the quotient, obtaining the remainder.
5. Add 1 to the remainder from step 4 and use this as a subscript to look up the corresponding valid character from the table built in step 3. Place this character in the formatted character position.

When the binary field in step **4** above, has undergone a number of division calculations equal to the number of ciphertext character positions it represents, logic switches to the next binary field to the right - this switch ensures the quotient will never be less than the radix.

Note: For decryption, an alternative to storing the original ciphertext in the application would be to store the residual binary field values. The ciphertext could then be recovered by running the method in reverse, multiplying by the radix and adding the remainder.

Example:

The requirement is to represent an encrypted 7 byte Canadian post code in the form: "ANA NAN" where "A" is any upper case or lower case alphabetic character, "N" is a numeric digit from 0 to 9 and the space in the middle must be a space.

As the AES ciphertext is hexadecimal: "**`e194afa36f35ca`**", this can be broken down into the following binary fields:

	PIC 9(8) BINARY	PIC 9(4) BINARY	PIC 9(4) BINARY
Hex:	`e194afa3`	`6f35`	`00ca`
Dec:	3784617891	28469	202

Table for "A":

A	B	C	D	E	F	G	H	I	J	K	L	M
N	O	P	O	R	S	T	U	V	W	X	Y	Z
a	b	c	d	e	f	g	h	i	i	k	l	m
n	o	p	q	r	s	t	u	v	w	x	y	z

Table for "N":

1	2	3	4	5	6	7	8	9	0

Table for space:

space

Each character position is then processed as follows**:**

Position	Table	Radix	BINARY before	BINARY after	Rema-inder	Subscript	Output
1	A	52	3784617891	72781113	15	16	P
2	N	10	72781113	7278111	3	4	4
3	A	52	7278111	139963	35	36	j
4	space	1	139963	139963	0	1	space
Switch to next binary field							
5	N	10	28469	2846	9	10	0
6	A	52	2846	54	38	39	m
Switch to next binary field							
7	N	10	202	20	2	3	3

Therefore, ciphertext "**`e194afa36f35ca`**" can be stored as post code: "P4j 0m3".

Note: The above method and example assume the data is held on a big endian platform. For little endian platforms the binary character positions will need to be reversed.

Feistel Networks

Feistel Network[14] designs produce format-preserved ciphertexts by repeatedly encrypting part of the plaintext and modulus adding the result to the unencrypted part of the plaintext. Decryption is achieved by running the network in reverse, modulus subtracting the encrypted result. Both processes are performed in a series of "rounds".

In 2013 the NIST issued Special Publication 800-38G[15] formalizing the use of Feistel Networks with AES to produce secure format-preserved ciphertexts. Three specific algorithms were defined: FF1, FF2 and FF3; requiring 10, 10 and 8 "rounds" respectively. *The FF2 algorithm was later dropped from the specification.*

The ability to produce consistent or inconsistent format-preserved ciphertexts for a given plaintext and key is provided by the use of a "Tweak". Tweaks are similar to Initialization Vectors (IV's) - see **Appendix B: Initialization Vectors**.

Consistent/inconsistent ciphertexts are particularly relevant to format-preserved encryption because of the possibility of ciphertext "*collisions*" (when two different plaintexts produce the same ciphertext) when producing inconsistent ciphertexts. The decision of whether to use consistent or inconsistent ciphertexts should made based on the security requirements of the data involved, not the potential inconvenience of ciphertext collisions – see **Points to Consider** section.

Advantages:

- ✓ Ciphertexts consist of characters from an application defined alphabet.
- ✓ Ciphertext lengths are equal to the plaintext lengths.
- ✓ Ciphertext "*collisions*" can be avoided if consistent ciphertexts are used.

Disadvantages:

- ✗ Additional processing requirement due to repeated "rounds" of encryption.
- ✗ Ciphertext "*collisions*" may be unavoidable if inconsistent ciphertexts are required for security reasons.
- ✗ Plaintext lengths must be greater than one character, preventing format-preservation at the character level.

The final point in the list above relates to fields like post codes or calendar dates where the validity of the field depends on more than just being entirely alphabetic or numeric. This difficulty can be overcome with a little application code before and after encryption/decryption:

Example 1:

The requirement is to format-preserve a 7 character Canadian post code in the form: "ANA NAN" where "A" is any upper case alphabetic, "N" is a numeric and " " is space.

The application reorders the plaintext characters into two, three byte strings: "AAA" and "NNN". Upper case, format-preserving encryption is then performed on the "AAA" string and numeric format-preserving encryption is performed on the "NNN" string. Finally, the encrypted characters are reordered back to their initial positions with the space inserted in position 4.

Decryption would run the same process in reverse.

Example 2:

The requirement is to format-preserve an 8 character date of birth in the form: "YYYYMMDD" where "YYYY" is the year, "MM" is a valid month number and "DD" is a valid calendar day within the month. All date of births are before 01/01/2015.

The application performs the following steps:

1. Convert the date of birth to a 5 digit integer equal to the number of days since 1900, using a COBOL FUNCTION or standard date routine.
2. Convert the 5 digit integer to a 6 digit base 6 number by dividing by 6, five times, placing each remainder in the next available right-most position. The final quotient is then placed in the left-most position.
3. Using numeric, format-preserving encryption, encrypt the 6 digit base 6 number using an alphabet of only 0 through 5.
4. Convert the 6 character ciphertext back to a base 10 integer by calculating the sum of: (7,776 x 1st digit, 1,296 x 2nd digit, 216 x 3rd digit, 36 x 4th digit, 6 x 5th digit, 1 x 6th digit).
5. Convert the 5 digit base 10 integer back to an encrypted date of birth after 1900, using a COBOL FUNCTION or site standard date routine.
6. If the date of birth from step **5** is > 20150000, repeat from step **3** using the previous step **3** ciphertext as the new plaintext.

Step	Process	Value
	Date of birth:	19600101
1	Number of days since 1900:	21914
2	Number of days in base 6:	245242
3	Number of days encrypted:	543210
4	Encrypted days in base 10:	44790
5	Encrypted date of birth:	20220819
6	Go to Step 3	
3	Number of days encrypted:	012345
4	Encrypted days in base 10:	1865
5	Encrypted date of birth:	19050209
6	Done	

Decryption runs the same process in the same sequence, including the step **6** check; except step **3** would *decrypt* the input ciphertext.

Note: The method above converts the number of days to base 6 and encrypts in base 6, purely to minimize the chance of having to repeat step 3 encryptions (approx 10%). If efficiency is not an issue, steps **2** and **4** can be skipped and a base 10 format-preserved numeric encryption can take place in step **3**, giving a 60% chance of having to repeat from step **3**.

Example 3:

The requirement is to encrypt credit card start-dates and expiry-dates as numeric characters in "MMYY" format, where no start-date is greater than its expiry-date.

The application builds a table consisting of pairs of valid start and expiry dates. If one list is longer than the other, place zeroes or some other invalid value in the shorter list entries. Each pair is then assigned a different alphanumeric character - in the 44 entry example below, "A" through "Z" followed by "a" through "r":

START-DATES	EXPIRY-DATES	Alphanumeric
0112	0515	A
0212	0615	B
0312	0715	C
0412 through 0315	0815 through 0718	D through Z & a through m
0415	0818	n
0515	0918	o
0000	1018	p
0000	1118	q
0000	1218	r

When a start/expiry date pair needs to be encrypted, the application performs the following steps:

1. Look up the start-date under the START-DATES column in the table and place the corresponding alphanumeric character in the first plaintext position.
2. Look up the expiry-date under the EXPIRY-DATES column in the table and place the corresponding alphanumeric character in the second plaintext position.
3. Encrypt the two character plaintext using format-preservation with an alphanumeric alphabet consisting of "A" through "Z" and "a" through "r".
4. Use the first character of the ciphertext to look up an encrypted start-date in the table.
5. Use the second character of the ciphertext to look up an encrypted expiry-date in the table.
6. If steps **4** or **5** result in an invalid date (zeroes), repeat from step **3** using the previous step **3** ciphertext as the new plaintext.

Decryption runs the same process in the same sequence, using the same table, including the step **6** check; except step **3** would *decrypt* the input ciphertext.

Binary Banks

Binary banks, ideal for high-volume transaction throughput, provide an alternative to real-time encryption using the AES cipher. The concept relies on the creation of a secretly held reserve of pseudo-random binary bytes to be combined with confidential data in an *exclusive or* binary operation. The aim is to create a One-time Pad[17], sometimes called a Vernam Cipher[18].

Advantages:

- ✓ Fast and efficient.

Disadvantages:

- ✗ The application must manage the used byte allocations within the bank.
- ✗ Possibility of ciphertext "*collisions*" (when two different plaintexts produce the same ciphertext) – this may, or may not be important to the application.
- ✗ Secure storage area required to hold the binary bank.

Method:

This approach requires the generation of an array, or bank, of pseudo-random binary data, held in a secured region of the application. This bank can be created by passing an initial, unique plaintext through the AES encryption process running in **Counter** mode or any other confidentiality mode (see **Appendix A: Confidentiality Modes**). The output is not only copied into the start of the binary bank but also used as the next plaintext input to AES encryption. This process is repeated until the binary bank is deemed large enough for all the confidential data in the application. *The bank can be extended at a later date, if necessary, by using the last output string in the bank as the next plaintext input to restart the binary bank generation process.*

When an application field needs to be encrypted, its contents are exclusively allocated a matching number of bytes from the binary bank. These bytes are then *exclusive or'd* (**XOR'd**) (see **Appendix E: Exclusive Or**) against the field content to produce the ciphertext for the field.

Decryption is just as simple. The ciphertext is *exclusive or'd* (**XOR'd**) against the same binary bank bytes, allocated in the encryption procedure. This returns the original plaintext.

The implementation of a binary bank solution requires that the application manages the allocations of binary bank pseudo-random bytes to their corresponding application fields, ensuring no binary bank bytes are allocated to more than one field, unless consistent ciphertexts are required - see **Points to Consider**. The application will also be required to monitor binary bank usage, retiring bytes when no longer required and extending the bank when necessary.

Note: An alternative to encrypting using *exclusive or* logic would be to add the numeric value of each binary bank character position to each plaintext character, using a modulus equal to the character position radix. For example, given a numeric plaintext character of "7" and a binary bank character value of 255, the resulting ciphertext character would be "2" (7 + 255 = 262, modulo 10 = 2). This technique has the added advantage of producing a format-preserved ciphertext. Decryption is achieved by modulo subtraction of the value of each binary bank character position.

Conclusion

We hope this document has proved the suitability of securing sensitive data using AES encryption software, written in COBOL. We have endeavoured to highlight any potential difficulties involved and to provide solutions to these difficulties.

To recap on the main points:

- Software encryption written in COBOL is a practical, simple and efficient option.
- AES is the primary secure global standard for symmetric-key encryption.
- Under AES, the application only needs to keep the encryption key secret.
- Any AES encryption product should support all confidentiality modes and key lengths.
- If pad characters are required, they should be as defined in Public-Key Cryptography Standards (PKCS#7)[19]
- If ciphertext is to be transmitted or stored in numeric or alphanumeric fields, it will need to be format-preserved or converted to hexadecimal or Base64.
- Storage areas containing decrypted plaintexts or encryption keys should be wiped clean after the encryption/decryption process has completed.

Fortunately, the Redvers Encryption Module[20] is an NIST validated[21], AES software product, written in COBOL and designed for COBOL applications. It supports the primary confidentiality modes (ECB, CBC, CFB, OFB and CTR), Fixed Format encryption and MAC / CCM modes for encryption based hash and authentication. All key lengths (128, 192 or 256 bit) and pad character methods are also supported and it includes a "clean storage" function to protect plaintexts and keys.

You can download a free, no obligation, 30 day trial of the Redvers Encryption Module from:
https://www.redversconsulting.com/data_encryption_free_trial.php

If you have any questions you'd like to ask us, please use our website Contact page at:
https://www.redversconsulting.com/contact.php

Appendix A: Confidentiality Modes

NIST Special Publication 800-38A[22] provides a concise description of each of the five AES confidentiality modes. The decision of which mode to use will depend on a combination of corporate policy, external providers and system requirements. To assist with this decision, the following sections highlight how the various confidentiality modes can be best utilized within a COBOL application environment.

Electronic Code Book (ECB)

Apart from the fact that this mode doesn't require an initialization vector or counter, the most important aspect of **Electronic Code Book** mode is that it will always produce the same ciphertext for a given plaintext and encryption key. Although this may not make it the most secure mode available, a consistent ciphertext will be essential for matching other encrypted values within the application.

Another aspect of **Electronic Code Book** mode is that it is the only confidentiality mode that guarantees a unique ciphertext for a given plaintext and key, eliminating the possibility of ciphertext "*collisions*" (when two different plaintexts produce the same ciphertext when encrypted). This may, or may not be important to the application.

In order to decrypt ciphertext created in **Electronic Code Book** mode, the AES inverse-cipher must be used. This inverse-cipher involves a slightly greater computer processing overhead than the forward cipher, due to the more complex binary arithmetic involved. **Cipher Feedback**, **Output Feedback** and **Counter** modes avoid this overhead by using the forward cipher for encryption and decryption.

Another potential drawback of **Electronic Code Book** mode is that it passes the plaintext and ciphertext through the AES cipher or inverse-cipher in complete blocks of 16 bytes. This requires the insertion of 1 through 16 pad characters (see **Appendix D: Pad Characters**) at the right-hand end of the plaintext when using this mode (even if the plaintext is a multiple of 16 bytes long). This expanded plaintext therefore produces an expanded ciphertext, which will always be longer than the original plaintext - see the sections within **Consistent Ciphertexts** if this is a problem.

The way that **Electronic Code Book** mode independently processes plaintext or ciphertext in discreet blocks of 16 bytes means that blocks can be simultaneously encrypted and decrypted using parallel processing. The potential downside is that blocks of 16 bytes could be removed from the ciphertext by an adversary without causing disruption to the legitimate decryption process. The other confidentiality modes do not have this weakness due to the way they chain blocks from one cipher event to the next.

Cipher Block Chaining (CBC)

Cipher Block Chaining requires an unpredictable initialization vector (IV) (see **Appendix B: Initialization Vectors**) for encryption and decryption. The use of the IV produces a different ciphertext for every plaintext.

As with **Electronic Code Book**, **Cipher Block Chaining** uses the AES inverse-cipher to decrypt ciphertext. This involves a slightly greater computer processing overhead than modes that use the forward cipher for decryption.

Also in keeping with **Electronic Code Book**, **Cipher Block Chaining** passes plaintext and ciphertext through the AES cipher or inverse-cipher in complete blocks of 16 bytes. This requires the addition of pad characters (see **Appendix D: Pad Characters**) and produces an expanded ciphertext - see the sections within **Consistent Ciphertexts** if this is a problem.

Cipher Feedback (CFB)

Cipher Feedback mode comes in four flavors: 1-bit mode, 8-bit mode, 64-bit mode, or 128-bit mode. The bit number relates to the quantity of encrypted bits used from each 128 bit AES cipher output block. All remaining bits are discarded. Therefore, given a plaintext length of 16 bytes (128 bits):

- 1-bit mode will require 128 AES cipher executions to encrypt/decrypt.
- 8-bit mode will require 16 AES cipher executions to encrypt/decrypt.
- 64-bit mode will require 2 AES cipher executions to encrypt/decrypt.
- 128-bit mode will require 1 AES cipher execution to encrypt/decrypt.

Practical applications for 1-bit mode will be rare due to the huge processing overhead but if only occasional encryption is required, or for the encryption of specific bit positions in a plaintext string, this mode could be chosen.

8-bit mode runs the AES cipher at the character level, avoiding the need for pad characters (see **Appendix D: Pad Characters**) and the corresponding ciphertext expansion. The overhead in executing the cipher once for every plaintext character is the only fundamental drawback.

64-bit and 128-bit modes are obviously faster than 1-bit and 8-bit modes but these modes will require the addition of pad characters and will therefore incur ciphertext expansion. However, 64-bit mode need only add a maximum of 8 pad characters to complete a plaintext half-block.

Like **Cipher Block Chaining**, **Cipher Feedback** mode requires an unpredictable initialization vector (IV) (see **Appendix B: Initialization Vectors**) for encryption and decryption. The use of the IV produces a different ciphertext for every plaintext.

Unlike **Cipher Block Chaining**, **Cipher Feedback** mode uses the more efficient AES forward cipher to encrypt and decrypt plaintexts.

Note: Although NIST Special Publication 800-38A[22] Appendix A states: "the plaintext must be a sequence of one or more complete data blocks (or, for CFB mode, data segments)", 64-bit and 128-bit CFB mode doesn't actually need a complete final data segment to function. The final operation in CFB encryption and decryption is an *exclusive or* (see **Appendix E: Exclusive Or**) between the AES cipher output and the plaintext or ciphertext. The addition of some simple logic surrounding the AES cipher could limit the *exclusive or* to the number of plaintext or ciphertext characters in the final segment, without the need for pad characters.

Output Feedback (OFB)

Output Feedback mode requires a unique initialization vector to produce a different ciphertext for every plaintext (see **Appendix B: Initialization Vectors**). This mode doesn't require the addition of pad characters and so produces a ciphertext, the same length as the plaintext. It also uses the more efficient forward cipher for both encryption and decryption.

This mode is frequently chosen for COBOL applications and is shown in more detail in the **Using OFB Mode** section.

Counter (CTR)

Counter mode offers COBOL applications a good alternative to **Output Feedback** mode. It requires a counter block (see **Appendix C: Counters**) instead of an initialization vector to produce a different ciphertext for every plaintext. Like **Output Feedback** mode, this mode doesn't require the addition of pad characters and it uses the more efficient forward cipher for both encryption and decryption.

Appendix B: Initialization Vectors

Initialization Vectors (IV's) weren't invented just to make life difficult for applications designers. They enhance the encryption process by enabling applications to generate inconsistent ciphertexts and they are vital for the production of ciphertext lengths equal to their corresponding plaintext lengths – particularly important for COBOL applications, tied to fixed length field formats.

What are IV's?

- IV's are a 16 byte block of data, required for **CBC**, **CFB** and **OFB** mode encryption and decryption.
- Only one IV is required for each encryption or decryption, regardless of the plaintext/ciphertext length.
- An IV has no format. It can be comprised of alphanumeric and/or binary characters in any order.
- The IV used for decryption must be the same as the one used for encryption.
- An IV doesn't need to be kept secret from the outside world.
- Every IV must be unique (a nonce) for every encryption operation, for a given key. *This must be true across all applications using the same key.*
- IV's used for **CBC** and **CFB** mode encryption must not be predictable. i.e. a potential intruder must not be able to generate multiple potential IV's as part of a Brute-force attack[11].

The generation of a unique IV doesn't usually present a problem for COBOL applications. Fields selected for encryption will usually be associated with sufficient key information to uniquely identify their location within the application database or file index. The concatenation of these key fields will produce a string, uniquely identifying the field location. Add to this a date-time or sequence number for the encryption event and you have everything you need to create a safe, unique IV.

Fields that make up the IV can be in any order and any format. Therefore, if the same key details are associated with more than one encryption, a resequencing or reformat of the fields involved is all that's necessary to produce another unique IV for additional encryptions. However, the decrypting application will need to use the same field sequence and formats to derive the correct IV for successful decryption.

If the total length of all the fields required to produce a unique IV comes to more than 16 bytes, various methods can be used to reduce the length of the IV string:

- Compress numeric data by moving it to fields defined as packed (COMP-3) or binary (COMP).
- Multiply multiple weighted numeric values to produce a single, larger product value.
- Generate unique hash totals to represent alphanumeric strings.

If it isn't possible to create a unique IV from non-encrypted key fields or if the decrypting application doesn't have access to the fields and information that make up the IV, an IV can be built from another source and sent, unencrypted, to the decryption application using another communication route or at the start of the ciphertext (IV's don't need to be kept secret).

Why must IV's be Unique?

There are several reasons why each IV must be unique for each OFB encryption. These reasons relate to the fact that, within the encryption routine, it's the IV that is actually passed through the encryption cipher logic, not the application plaintext.

As mentioned in the **Executive Summary**, AES encryption is the process of combining data that needs to be kept secret, with a pseudo-random bit string. In OFB mode, it is the encrypted IV that provides this pseudo-random string, which is then combined with the plaintext, using *exclusive or* (**XOR**) logic (see **Appendix E: Exclusive Or**), to produce the Ciphertext - see diagram in the **Using OFB Mode** section. Therefore, if the same IV is input to the encryption cipher, using the same key, the encrypted IV will also be the same.

Looking at the consequences of using duplicate IV's in more detail: the laws of binary mathematics mean that if two different, publicly known ciphertexts, produced from the same IV, were to be **XOR**'d together the result would be a binary map of the precise difference between the two original plaintexts - removing the effect of the IV, cipher and key! From this point, it isn't difficult to derive all confidential plaintext, passed to the encryption processes that used the same IV.

To prove the point above, we can look at what happens when two single characters, "A" and "Z", are encrypted using the same IV (simplified to one character). First, let's assume the duplicated IV, passed through the encryption cipher produced an encrypted IV binary string of "10101010". Using EBCDIC representation, the letter "A" is binary "11000001" and the letter "Z" is binary "11101001":

At the end of the encryption process, the encrypted IV is **XOR**'d with the plaintext binary "A" giving ciphertext-A:

```
Encrypted IV: 1 0 1 0 1 0 1 0
Plaintext-A:  1 1 0 0 0 0 0 1
              ---------------
Ciphertext-A: 0 1 1 0 1 0 1 1
              ---------------
```

In another encryption process, the same encrypted IV is **XOR**'d with plaintext binary "Z" giving ciphertext-Z:

```
Encrypted IV: 1 0 1 0 1 0 1 0
Plaintext-Z:  1 1 1 0 1 0 0 1
              ---------------
Ciphertext-Z: 0 1 0 0 0 0 1 1
              ---------------
```

Now, let's **XOR** publicly known Ciphertext-A with publicly known Ciphertext-Z:

```
Ciphertext-A: 0 1 1 0 1 0 1 1
Ciphertext-Z: 0 1 0 0 0 0 1 1
              ---------------
XOR Result:   0 0 1 0 1 0 0 0
              ---------------
```

The result pinpoints the plaintext bit positions (3rd and 5th from the left) that must be changed to switch plaintext "A" to plaintext "Z" or plaintext "Z" to plaintext "A".

Appendix C: Counters

Counter blocks are used in **Counter (CTR)** mode encryption and decryption in place of an Initialization Vector (IV) parameter (see **Appendix B: Initialization Vectors**). Like IV's, counters must be a unique 16 byte block. They can be in any format and they don't need to be kept secret. The same counter block used for encryption, is required for decryption.

Unlike IV's, a different counter block is required for each 16 bytes of input (plaintext or ciphertext). Good encryption software like the Redvers Encryption Module[20] will increment the first counter block passed to the subroutine by binary 1, internally, for each additional 16 bytes of input. The last counter block + binary 1, is then returned to the application for use by the next Counter mode encryption.

Another difference between using a counter and IV is that in order to ensure uniqueness of counter blocks, their use must be managed centrally so every encryption uses a different range of counter values. This may require single streaming applications that use the same encryption key.

As the first counter block used for encryption is also required for decryption, it must be passed to the decrypting application. This can either be done, by placing it at the start of the ciphertext or by using another communication route. Alternatively, synchronised counters may be maintained within both the encryption and decryption applications.

A common deployment of Counter mode encryption is for the generation of a store of pseudo-random binary strings, ready to be *exclusive or'd* (see **Appendix E: Exclusive Or**) with plaintext for rapid encryption and decryption at busy times – see **Binary Banks**.

Appendix D: Pad Characters

In order to encrypt using **Electronic Code Book (ECB)**, **Cipher Block Chaining (CBC)** and sometimes **Cipher Feedback (CFB)** confidentiality modes (see **Appendix A: Confidentiality Modes**), pad characters are added to the right of the original plaintext to make up a complete number of 16 byte blocks. These additional pad characters are removed at the end of the decryption procedure so that the plaintext length is restored.

If the plaintext length is already an exact multiple of 16, another block of 16 pad characters are added so that the decryption process can correctly identify the end of the plaintext. Without the guarantee that the output from the decryption cipher terminates with a pad character, it would be impossible to tell a pad character from the final data byte.

The addition and removal of pad characters occurs within the encryption/decryption subroutines so pad character creation and destruction isn't usually an issue for applications. However, COBOL application designers should be aware of inappropriate padding methods.

There are five standard padding methods used by encryption software today. The table below shows how a 12 character string of "HELLO WORLD!" would be padded to a 16 byte input block (in EBCDIC):

Method 1: (PKCS#5, PKCS#7 & RFC3369)	Every pad character is the number of pad characters added (in hexadecimal).	HELLO WORLD!.... ccddd4edddc50000 85336066934A4444
Method 2:	The leftmost pad character is hex "80", followed by hex "00" characters (if any).	HELLO WORLD!.... ccddd4edddc58000 85336066934A0000
Method 3:	The rightmost pad character is the number of pad characters added (in hexadecimal), preceded with hex "00" characters (if any).	HELLO WORLD!.... ccddd4edddc50000 85336066934A0004
Method 4:	Every pad character is hex "00".	HELLO WORLD!.... ccddd4edddc50000 85336066934A0000
Method 5:	Every pad character is a space (hex "40" in EBCDIC, hex "20" in ASCII).	HELLO WORLD!.... ccddd4edddc54444 85336066934A0000

The table above shows how methods 1 and 3 safely indicate the exact number of pad characters to be removed, by an inspection of the byte in position 16. However, additional characters could be removed in methods 2, 4 and 5 if the plaintext happened to end in hex "80", LOW-VALUES or SPACES – common values in COBOL applications.

Note: The Redvers Encryption Module[20] uses Method 1 - Public-Key Cryptography Standards (PKCS#7)[19] padding for ECB, CBC and CFB mode encryption. Decryption will accept any of the five standard methods.

Appendix E: Exclusive Or (XOR)

The eXclusive Or (**XOR**) process compares each of the 8 bits within a byte from one string, with the corresponding 8 bits within a byte from another string. If both bits are zero or both bits are one, the resulting bit will be set to zero. Otherwise, the resulting bit will be one.

E.g.:

```
Byte1:  1 0 0 1 1 0 1 1
Byte2:  0 1 0 1 0 0 1 0
        ---------------
Result: 1 1 0 0 1 0 0 1
        ---------------
```

The effect of the **XOR** function means that any plaintext character can be converted directly to a ciphertext character by **XOR**'ing it with a string of 8 pseudo-random bits. Continue this process for all the characters in a plaintext string and you have a complete ciphertext. Pseudo-random bits can be generated from tossing coins, an AES encryption subroutine or from some other reliably random source.

Decryption is achieved by **XOR**'ing the ciphertext with the same pseudo-random bits used for the encryption. The result is the original plaintext.

E.g.: (using the characters from above):

```
Result: 1 1 0 0 1 0 0 1
Byte2:  0 1 0 1 0 0 1 0
        ---------------
Byte1:  1 0 0 1 1 0 1 1
        ---------------
```

The **XOR** process is quite simple and very fast if optimised correctly. Redvers Consulting offers a free, optimised, downloadable COBOL **XOR** subroutine: RCNOTOR[24].

Appendix F: References

[1] National Institute of Standards and Technology: https://www.nist.gov/index.html

[2] FIPS PUB 197: https://csrc.nist.gov/publications/detail/fips/197/final

[3] National Security Agency: https://www.nsa.gov/

[5] PCI DSS: https://www.pcisecuritystandards.org/pci_security/

[8] W3C Extensible Markup Language (XML): https://www.w3.org/TR/REC-xml/

[9] Base64: https://en.wikipedia.org/wiki/Base64

[10] RCBINB64: https://www.cobol.org.uk

[11] Brute-force attack: https://en.wikipedia.org/wiki/Brute_force_attack

[12] Ciphertext Stealing: https://en.wikipedia.org/wiki/Ciphertext_stealing

[13] Redvers Hashing Algorithm: https://www.redversconsulting.com/hashing_algorithm.php

[14] Feistel Network: https://en.wikipedia.org/wiki/Feistel_cipher

[15] Special Pub 800-38G: https://csrc.nist.gov/publications/detail/sp/800-38g/final

[17] One-time pad: https://en.wikipedia.org/wiki/One-time_pad

[18] Vernam Cipher: https://www.pro-technix.com/information/crypto/pages/vernam_base.html

[19] Public-Key Cryptography Std (PKCS#7): https://tools.ietf.org/html/rfc2315

[20] Redvers Encryption Module: https://www.redversconsulting.com/data_encryption.php

[21] NIST: https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program/

[22] Special Pub 800-38A: https://csrc.nist.gov/publications/detail/sp/800-38a/final

[24] RCNOTOR: https://www.cobol.org.uk

Appendix G: About Redvers Consulting

Redvers Consulting provide niche software products for the integration, modernization and security of COBOL applications. Our clients are primarily large financial institutions in Europe and North America, although we also have customers in many other business and geographical areas.

Our ability to deliver software in COBOL source code form, gives customers reliable, efficient and perfectly integrated solutions to business needs. Source code distribution also means our software will run on all hardware platforms and operating systems: *EBCDIC, ASCII, big endian or little endian*.

Redvers Consulting have received many business awards over the years, including winning the **Best use of Technology** category in the Thames Gateway Business Awards - see logos below. We are also business partners with **IBM**, **Micro Focus** and **Fujitsu**.

Our client list includes:

Agora (FR)
ANZ (AUS)
BAE Systems (USA)
Canada Life Assurance (UK)
Deutsche Bank (USA)
Deutsche Rentenversicherung Bund (DE)
FirstBank (USA)
Fiserv (USA)
GMAC Insurance (USA)
Hanesbrands (USA)
John Deere (USA)
LBS / Finanz Informatik (DE)
J P Morgan (USA)
Oppenheimer (USA)
Pacific Gas (USA)
Network Rail (UK)
R+V Allgemeine Versicherung (DE)
Sasktel (CAN)
SEB (DE)
Standard Life Assurance (UK)
Suncorp (AUS)
SunGard / FIS (USA)
WorkSafeBC (CAN)
Zurich Insurance (UK & CHE)

Contact: https://www.redversconsulting.com/contact.php

Development Office:

Redvers Consulting Ltd
16-18 Woodford Road,
London E7 0HA,
UK

Tel: +44 (0)208 522 7404

Accounts Office:

Redvers Consulting Ltd
1st Floor, 48 Dangan Rd,
London E11 2RF,
UK

Tel: +44 (0)870 922 0633

German Office:

Redvers Consulting Ltd
Scharfeneckweg 2,
50739 Köln,
Germany

Tel: +49 (0)221 1704 9000