

Redvers Consulting Ltd

White Paper

**Schützen Sie
vertrauliche Daten mit
AES Verschlüsselung -
geschrieben in COBOL
entwickelt für COBOL**



Inhaltsverzeichnis

Zusammenfassung.....	3
Alternativen zur Nutzung von COBOL AES Software	4
COBOL Verschlüsselung in der täglichen Anwendung.....	6
AES Verschlüsselungsmethodik	8
Eigenschaften von AES	8
Relevante Auswahlkriterien	8
Konsistente Geheimtexte.....	12
Die Verwendung des ECB Modus	12
Die Entfernung des Füllzeichenblocks	14
Gestohlener Geheimtext	16
Klartext anderer Längen	19
Inkonsistente Geheimtexte	21
Die Nutzung des OFB-Modus	21
Verschlüsselung ohne Entschlüsselung	23
Einwegverschlüsselung	23
Hash-Summe	24
Geheimtexte mit einem erhalten gebliebenem Format.....	25
Geheimtext-Sortierung	25
Radix-Aufteilung	28
Feistel Netzwerke.....	30
Binäre Speicherbänke	34
Zusammenfassung.....	36
Anhang A: Vertraulichkeitsmodi.....	37
Electronic Code Book (ECB).....	37
Cipher Block Chaining (CBC)	37
Cipher Feedback (CFB)	38
Output Feedback (OFB)	39
Counter (CTR)	39
Anhang B: Initialisierungsvektoren	40
Anhang C: Zähler	43
Anhang D: Füllzeichen.....	44
Anhang E: Exclusives Oder (XOR)	45
Anhang F: Referenzen.....	46
Anhang G: Über Redvers Consulting	47

Zusammenfassung

Der Sinn dieses Aufsatzes ist es nicht, Sie davon zu überzeugen, Ihre Daten zu verschlüsseln. Wenn Sie diese Zeilen lesen, wissen Sie bereits, dass Sie vertrauliche Daten sicher aufbewahren, also verschlüsseln müssen. Dieser Aufsatz hilft Ihnen dabei, den Verschlüsselungsprozess einfach und sicher in Ihre COBOL-Anwendungen zu integrieren, ohne dass sie kostspielige Dienstleistungen von Verschlüsselungsspezialisten einkaufen müssen.

Grundsätzlich schützt die Verschlüsselung Daten, die geheim gehalten werden müssen, indem sie mit einer scheinbar zufälligen Bit-Folge kombiniert werden. Ein völlig ausreichender Zufallsstring könnte erzeugt werden, indem wir ein Zimmer voll mit Menschen packen, und sie anweisen, wiederholt eine Münze zu werfen, und dann die Ergebnisse zusammenfassen. Zum Glück gibt es arbeitsparende Erfindungen, die Computer genannt werden, und mit deren Hilfe man einen Pseudo-Zufallsstring generieren kann, und das viel zuverlässiger und zu viel geringeren Kosten.

Die sichere, zuverlässige Erzeugung solcher Pseudo-Zufallsstrings ist der Grund, wieso der *Advanced Encryption Standard* (AES) entwickelt wurde.

Was ist AES-Verschlüsselung?

AES-Verschlüsselung beruht auf dem Rijndael-Algorithmus, der von zwei belgischen Kryptografie-Experten entwickelt wurde: Vincent Rijmen und Joan Daemen.

Im November 2001 wählte das amerikanische [National Institute of Standards and Technology](#)^[1] (NIST) den Rijndael-Algorithmus als offiziellen Verschlüsselungsstandard, den Advanced Encryption Standard (AES) Algorithmus ([FIPS PUB 197](#)^[2]) für die Verschlüsselungsbedarf der amerikanischen Bundesregierung aus.

Dann wählte im Juni 2003 die [National Security Agency](#)^[3] (NSA) die AES-Verschlüsselung aus, mit dem Hinweis „alle Schlüssellängen des AES-Algorithmus (also 128, 192 und 256) sind ausreichend, um vertrauliche Informationen bis zur Stufe GEHEIM zu sichern“ und „STRENG GEHEIME Informationen müssen eine Schlüssellänge von 192 oder 256 verwenden.“^[4]

AES eignet sich auch hervorragend für die Verarbeitung von Kredit- und Debitkartenzahlungen unter Verwendung des Datensicherheitsstandards PCI DSS ([Payment Card Industry Data Security Standard](#)^[5]).

Warum sollte man COBOL verwenden?

Da die meisten Systeme im Finanzwesen in COBOL^[6] geschrieben sind, liegt der Schwerpunkt dieses Aufsatzes bei der Installation von AES-Verschlüsselung bei COBOL-Anwendungsprogrammen.

Hinweis: Sie müssen nur grundlegendes Wissen über Verschlüsselung und COBOL besitzen, um die einzelnen Punkte, die in diesem Aufsatz aufgeführt sind, zu verstehen.

Alternativen zur Nutzung von COBOL AES Software

Hardware oder Software?

Hardware-basierte Verschlüsselung auf Zusatzgeräten:

- ✗ Die Anschaffung zusätzlicher Hardware ist nötig, was die Komplexität und die Wartungskosten erhöht.
- ✗ Wenn die Hardware ausfällt, kann das bedeuten, dass die Daten verschlüsselt sind und es keine Möglichkeit gibt, sie zu entschlüsseln.
- ✗ Alle Daten, die auf dem Gerät gespeichert werden, müssen verschlüsselt werden, nicht nur die sensiblen Felder.
- ✗ Alle Daten werden beim Lesevorgang entschlüsselt, auch wenn die Abfrage absolut trivial oder unsicher ist.
- ✗ Da der komplette Datenträger ver- und entschlüsselt wird, erhöht sich der Verarbeitungsaufwand, und damit steigen Kosten für Strom und Kühlung.
- ✓ Die Anwendungsprogramme müssen nicht verändert werden.

Softwareverschlüsselung, die durch Anwendungsprogramme initiiert wird:

- ✗ Der Programmiercode der Anwendungsprogramme muss geringfügig verändert werden.
- ✓ Nur die Daten, die der Geheimhaltung unterliegen, müssen verschlüsselt werden.
- ✓ Die meisten Anwendungsdaten bleiben da, wo sie vorher waren, und es kann ohne Veränderung auf sie zugegriffen werden.
- ✓ Es können unterschiedliche Verschlüsselungsmodi, Schlüssel und Schlüssellängen verwendet werden.
- ✓ Der Verschlüsselungsalgorithmus kann einfach verändert, d.h. die Sicherheit erhöht werden.
- ✓ Veränderungen der Verschlüsselungsparameter können inkrementel durchgeführt werden.

Welche Programmiersprache eignet sich am besten für die Verschlüsselung?

- ✗ **Assembler** ist oftmals eine effiziente Wahl, wenn es sich um kleinere Anwendungen handelt. Der AES-Algorithmus besteht jedoch aus einer langen, komplexen Abfolge mathematischer Operationen, was sich für Assembler nicht eignet. Ein optimiertes COBOL-Programm, das mit einem guten Compiler kompiliert wurde, läuft in den meisten Fällen schneller.
- ✗ **Java** benötigt zur Durchführung derselben logischen Operation etwa die zehnfache Rechenleistung, die COBOL brauchen würde ^[7]. Selbst wenn diese Arbeitslast an einen Spezialprozessor übergeben wird, sind die Gesamt-CPU-Zeit und auch die Verarbeitungszeit dennoch größer als unter COBOL.
- ✓ **COBOL**-Unterprogramme, die von einem COBOL-Anwendungsprogramm aufgerufen werden, führen zu der effizientesten und am besten kompatiblen

Lösung, unter Verwendung der geringsten CPU-Last und ohne der Notwendigkeit, neu geschultes Personal einzusetzen.

Standardprogramme oder individuell programmiert?

Da es sich beim AES-Algorithmus um eine Abfolge komplexer Datenverarbeitungsbefehle unter Verwendung binärer Mathematik handelt, die weltweit standardisiert ist, ist es sinnvoll, kommerziell erhältliche Ver- und Entschlüsselungsunterprogramme zu verwenden. Ein professioneller Softwarehersteller hat den Programmiercode optimiert und wird zukünftige Verbesserungen des AES-Algorithmus, falls diese anstehen sollten, unterstützen.

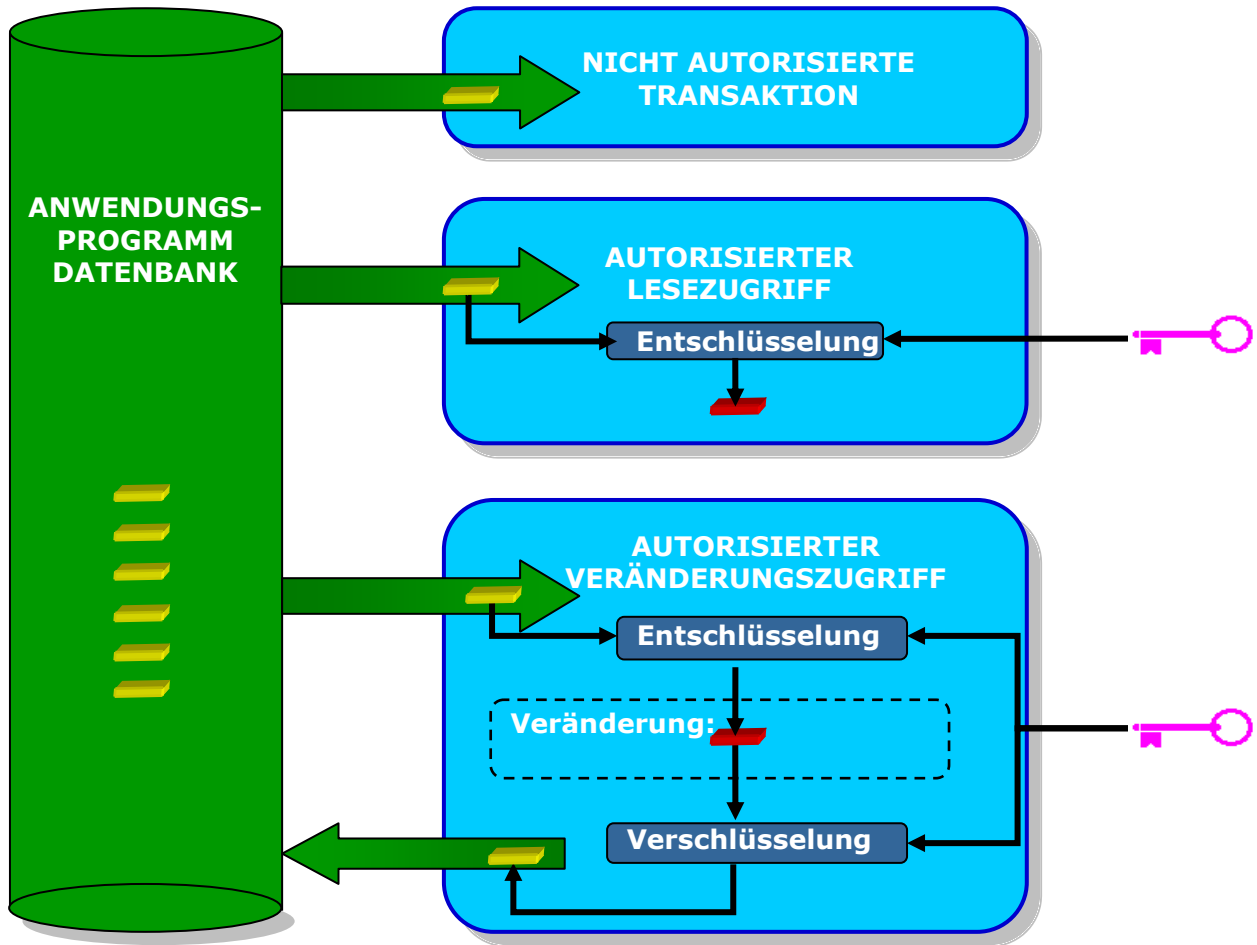
COBOL Verschlüsselung in der täglichen Anwendung



Wenn Sie sich für COBOL AES Verschlüsselung entscheiden, müssen Sie einmalig alle Felder, die geheimhaltungswürdige Daten (***Klartext***) enthalten, durch die entsprechenden verschlüsselten Felder (***Geheimtext***) ersetzen. Nachdem dieser Vorgang abgeschlossen ist, bleiben die Anwendungsdatenbanken und -dateien intakt und können genauso verwendet werden, wie es zuvor der Fall war. In den meisten Fällen führt die Entscheidung, vertrauliche Daten zu verschlüsseln zu keiner wie auch immer gearteten Notwendigkeit, ein Programm anzupassen.

Unsichere Transaktionen können weiterhin, wie bisher, auf Datensätze zugreifen, die verschlüsselte Felder enthalten; sie können lediglich den Inhalt der *Geheimtext*-Felder nicht lesen oder bearbeiten.

Wenn ein Anwendungsprogramm autorisiert ist, auf verschlüsselte Daten zuzugreifen, übergibt es den *Geheimtext* und den Schlüssel an ein Entschlüsselungsunterprogramm, das den lesbaren *Klartext* zurückgibt.

Wenn das autorisierte Anwendungsprogramm die verschlüsselten Daten verändern muss, verwendet es die übliche Entschlüsselungsmethode und ändert dann den *Klartext*. Der veränderte *Klartext* wird dann dem Verschlüsselungsprogramm übergeben (zusammen mit dem Schlüssel) und das neu ausgegebene *Geheimtext*-Feld wird in die Datenbank oder Datei zurückgeschrieben.



-  verschlüsselte, vertrauliche Daten (Geheimtext)
-  entschlüsselte, vertrauliche Daten (Klartext)

AES Verschlüsselungsmethodik

Eigenschaften von AES

- Verschlüsselung in Blöcken - jeweils 128 Bit (16 Zeichen) werden gemeinsam bearbeitet.
- Algorithmus mit symmetrischen Schlüsseln - zur Ver- und Entschlüsselung wird derselbe Schlüssel genutzt.
- Die Schlüssellänge kann 128, 192 oder 256 Bit betragen.
- Der Algorithmus geschieht aufgrund eines bekannten, logischen Prozesses - nur der verwendete Schlüssel muss geheim gehalten werden.

Der letzte Punkt in der Aufzählung oben ist für die Verwaltung von COBOL-Anwendungen besonders relevant. Das bedeutet nämlich, dass der Schlüssel zwar sicher aufbewahrt werden muss, der Quellcode der Software jedoch in den üblichen, frei zugänglichen Libraries aufbewahrt werden kann - was bei COBOL-Anwendungsprogrammen der Regelfall ist.

AES-Verschlüsselung ist keine Lösung, bei der jeder dasselbe Produkt auswählen muss. Es gibt mehrere, unterschiedliche Anwendungsszenarien, und weitere Eigenschaften, die dazu beitragen, dass sie sich nahtlos in viele unterschiedliche Softwareumgebungen einfügt. Die Auswahl des Verschlüsselungsmodus, der Schlüssellänge und der anderen Parameter hängt vom jeweiligen Anwendungsprogramm ab. Daher sollten die folgenden Punkte beachtet werden, bevor man ein AES-Projekt ins Auge fasst.

Relevante Auswahlkriterien

Sie können die nachfolgenden Fragen nutzen, um nur diejenigen Abschnitte dieses Dokuments auszuwählen, die auf Ihr Projekt zutreffen:

Sollte mein verschlüsselter Text im Ergebnis immer gleich sein, wenn ein bestimmter Text verschlüsselt wird?

Wenn ein verschlüsselter Text verwendet werden soll, um entsprechende verschlüsselte Werte innerhalb einer Anwendung darzustellen (Passwörter, Identifikationsnummern oder andere wichtige Felder), muss ein verschlüsselter Text für einen bestimmten zu verschlüsselnden Text immer denselben Wert annehmen - zu Details siehe den Abschnitt **Konsistente Geheimtexte**.

Um maximale Sicherheit zu erzielen, muss das Anwendungsprogramm jedoch einen unterschiedlichen verschlüsselten Text erzeugen, auch wenn er zu verschlüsselnde Text einen jeweils gleichen Wert hat. Wenn zum Beispiel die Ablaufdaten in einer Kreditkartenanwendung als konsistente verschlüsselte Texte erzeugt würden, würde es ausreichen, das richtige Ablaufdatum nur einer einzigen Kreditkarte zu wissen, um das Ablaufdatum aller Karten, die bis zum selben Monat gültig sind, zu enthüllen - zu Details siehe den Abschnitt **Inkonsistente Geheimtexte**.

Was macht man, wenn keine Notwendigkeit besteht, den verschlüsselten Text später wieder zu entschlüsseln?

Bei der Überprüfung einer Nutzerkennung, von Passwörtern oder PIN-Nummern ist es oft so, dass keine Entschlüsselung benötigt wird. Es kann sogar ausreichend sein, eine Standard-Hashroutine zu verwenden, anstelle von Verschlüsselung. Wenn die Eingabe nach der Verschlüsselung oder dem Hashprozess dem zuvor verschlüsselten oder gehashten Wert der Nutzerkennung, des Passwortes oder der PIN-Nummer entspricht, konnte die Eingabe verifiziert werden. Die Anwendung muss nicht die tatsächlich verwendete Nutzerkennung, das Passwort oder die verwendete PIN erfahren - siehe den Abschnitt [Verschlüsselung ohne Entschlüsselung](#), wenn Sie weitere Details erfahren möchten.

Welche Schlüssellänge sollte man verwenden?

Die Wahl der Schlüssellänge von 128, 192 oder 256 Bit hängt von der für die verarbeiteten Daten benötigten Sicherheitsstufe ab. 128 Bit wird für die meisten kommerziellen Anwendungen in der Regel als ausreichend angesehen.

Ein 192 Bit-Schlüssel benötigt etwa 20% höhere Rechenleistung als ein 128 Bit-Schlüssel und ein 256 Bit-Schlüssel benötigt eine ungefähr 40% höhere Rechenleistung als ein 128 Bit-Schlüssel.

Kann Geheimtext erzeugt werden, der später dann von einer externen AES-Verschlüsselungsroutine entschlüsselt wird (oder umgekehrt)?

Ja. Bei der AES-Verschlüsselung handelt es sich um einen globalen Standard, der unabhängig ist von Rechner oder Programmiersprache. Der verwendete Schlüssel, der Vertraulichkeitsmodus (siehe [Anhang A: Vertraulichkeitsmodi](#)) und, falls nötig, der Initialisierungsvektor (siehe [Anhang B: Initialisierungsvektoren](#)) müssen jedoch alle bei Ver- und Entschlüsselung übereinstimmen, daher muss man sich auf diese vorab mit dem Drittanbieter verständigen.

Kann Cipertext in einem XML-Dokument übertragen werden?

Nein - auch nicht innerhalb eines CDATA-Abschnitts. Geheimtext besteht aus scheinbar zufälligen Bitmustern, die zufällig jedes Zeichen, oder Zeichenfolge, darstellen könnten. So würde beispielsweise das „<“-Zeichen durchschnittlich einmal pro 256 Zeichen Geheimtext auftreten ($1:2^8$) und die Zeichenfolge „]]>“ würde durchschnittlich einmal pro 17 MB Zeichen Geheimtext auftreten ($1:2^{24}$). Außerdem gibt es bei XML nur einen [begrenzten Zeichensatz](#)^[8], so dass ungültige Zeichen, die im Geheimtext auftauchen, dazu führen, dass XML nicht „wohlgeformt“ ist.

Die beste Lösung für dieses Problem ist es, den Geheimtext ins [Base64](#)^[9] Format zu konvertieren (Zeichensatz von A-Z, a-z, 0-9, +, / und =). Das kann entweder innerhalb des AES-Verschlüsselungs-Unterprogramms durchgeführt werden, oder indem man ein separat erhältliches Softwaretool nutzt, um Geheimtext in Base64 umzuwandeln, beispielsweise [RCBINB64](#)^[10].

Die Umwandlung in Base64 führt notwendigerweise zu einem Geheimtext, der etwa 33% länger ist als zuvor.

Kann Geheimtext komprimiert werden?

Es kommt auf einen Versuch an. Da die Bitmuster im Geheimtext jedoch zufällig verteilt sind, ist es nicht unwahrscheinlich, dass die Länge des komprimierten Textes größer ist als die des unkomprimierten Textes. Allerdings ist es möglich, komprimierten Klartext zu verschlüsseln. Daher sollte folgende Verarbeitungsreihenfolge eingehalten werden:

komprimieren, verschlüsseln, → entschlüsseln, dekomprimieren.

Wie erreicht man ein vorgeschriebenes Format des Geheimtextes?

Geheimtext kann normalerweise ohne Probleme in alphanumerischen COBOL-Felder (`PICT X`) abgelegt werden. Wenn das Feld, in dem der Geheimtext gespeichert werden soll, jedoch numerisch definiert ist, oder wenn es einem spezifischen Format entsprechen muss, wird das zu Formatierungsproblemen führen (da Geheimtext aus einer zufälligen Abfolge von Bits besteht). Wenn Geheimtexte in Feldern abgespeichert werden sollen, die eingeschränkte Formate oder Werte besitzen, lesen Sie bitte den **Geheimtexte mit einem erhalten gebliebenem Format**.

Wie kann man große Datenmengen sehr schnell ver- und entschlüsseln?

Anwendungsprogramme, die große Datenmengen sehr schnell verarbeiten müssen, können möglicherweise an Leistungsgrenzen stoßen, wenn AES-Algorithmen in Stoßzeiten ausgeführt werden sollen. Um dieses Problem zu vermeiden, kann in Zeiten, in denen die Prozessorlast geringer ist, ein Vorrat an Pseudozufallsbinärstrings angelegt werden. Dieser Vorrat kann dann dazu verwendet werden, in Stoßzeiten vertrauliche Daten zu ver- und entschlüsseln, indem die *exclusive or* Logik verwendet wird (siehe **Anhang E: Exclusives Oder (XOR)**). Zu weiteren Details lesen Sie bitte den Abschnitt **Binäre Speicherbänke**.

Gibt es noch etwas, das man beachten sollte?

In der Anwendungssteuerung ist es von überragender Wichtigkeit, die zur Verschlüsselung verwendeten Schlüssel sicher aufzubewahren. Da jede Installation andere Eigenschaften hat und die Details der Aufbewahrung der Schlüssel geheimzuhalten sind, ist ein öffentlich frei verfügbares White Paper sicher nicht der richtige Ort, um zu beschreiben, wie mit den Schlüsseln umgegangen werden sollte.

Vor diesem Hintergrund soll hier dennoch darauf hingewiesen werden, dass die verwendeten Schlüssel wahrscheinlich einen Angriff eher standhalten können, wenn sie aus einem zufälligen Bitmuster bestehen, und nicht aus ausdrückbaren Zeichen. Das kann beispielsweise dadurch erreicht werden, indem man einen alphanumerischen String als Ausgangsbasis verwendet und diesen dann AES-verschlüsselt, wobei die Ausgabe als Schlüssel verwendet wird.

Es hat sich auch als vorteilhaft erwiesen, die verwendeten Felder im Anwendungsprogramm, die Schlüssel oder Klartext enthalten, direkt nach Beendigung der Ver- bzw. Entschlüsselung zu initialisieren. Gute Verschlüsselungssoftware sollte auch die Möglichkeit bieten, alle Speicherbereiche, die Schlüssel oder unverschlüsselte Daten enthalten können, zu initialisieren.

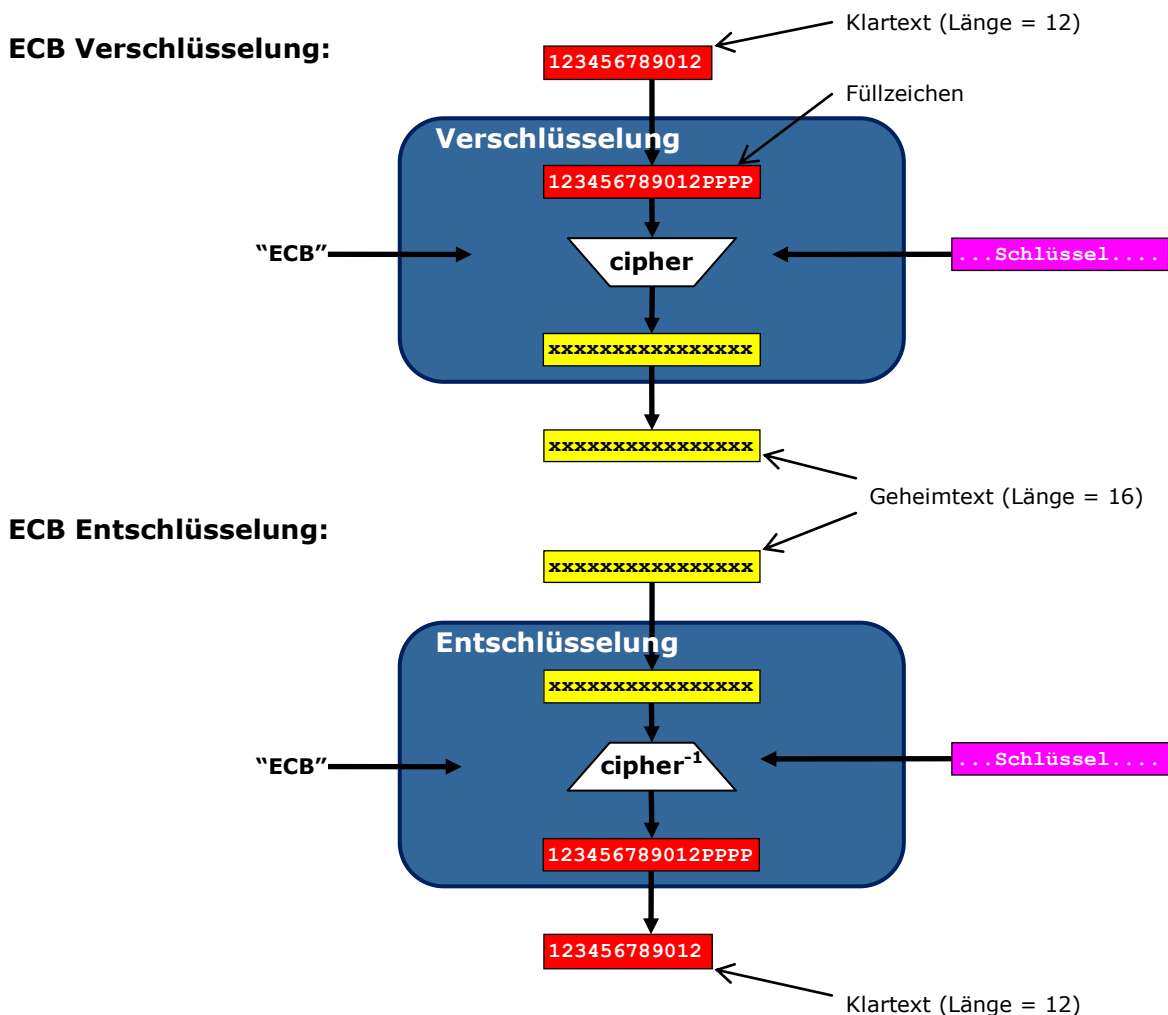
Falls ein Schlüssel unbrauchbar wird, weil er unberechtigten Personen zugänglich gemacht wurde, muss schnellstmöglich ein neuer Schlüssel erzeugt und alle verschlüsselten Felder neu verschlüsselt werden. Falls umständehalber keine hohe Dringlichkeit besteht, besteht die Möglichkeit, den Übergang zum neuen Schlüssel schrittweise durchzuführen, indem anhand des Datums/der Zeit des letzten Verschlüsselungsprozesses festgestellt wird, ob der alte oder der neue Schlüssel für die Entschlüsselung verwendet werden sollte, und dann, falls nötig, eine Neuverschlüsselung unter Verwendung des neuen Schlüssels durchzuführen.

Konsistente Geheimtexte

Wenn die AES-Verschlüsselung für einen vorgegebenen Klartext und Schlüssel einen konsistenten Geheimtext-String erzeugen soll, muss die AES-Verschlüsselung zusammen mit dem Modus Electronic Code Book (ECB) verwendet werden (siehe **Anhang A: Vertraulichkeitsmodi**).

Die Verwendung des ECB Modus

Um einen Klartext zu verschlüsseln, übergibt ein Anwendungsprogramm den Modus („ECB“), den Schlüssel und den Klartext an das Verschlüsselungsunterprogramm. Das Unterprogramm gibt den entsprechenden konsistenten Geheimtext zurück. Zur Entschlüsselung wird derselbe Modus und Schlüssel dem Entschlüsselungsunterprogramm übergeben, zusammen mit dem Geheimtext, und der Klartext kommt zurück. Unten ist das dargestellt:



Der wichtigste Nachteil bei der Verwendung des ECB Modus besteht darin, dass damit nie ein Geheimtext erzeugt werden kann, der dieselbe Länge wie der Klartext hat. Mindestens ein Füllzeichen (siehe **Anhang D: Füllzeichen**) muss immer rechts an den Klartext angehängt werden, damit das Entschlüsselungsprogramm die Länge des ursprünglichen Klartextes bestimmen kann. Außerdem muss der Klartext an das AES Verschlüsselungsprogramm immer in kompletten 16-Byte-Blöcken übergeben werden. Die Kombination dieser beiden Bedingungen bedeutet, dass

der Geheimtext immer zu einer Geheimtextlänge führt, die auf das nächste Vielfache von 16 verlängert wird (selbst wenn der Klartext schon eine Länge hat, die exakt ein Vielfaches von 16 ist).

Wenn eine Entschlüsselung benötigt wird, braucht man jedes einzelne Zeichen des verschlüsselten Textes als Eingabe für die Entschlüsselungsroutine, daher muss berücksichtigt werden, dass verschlüsselte Texte immer länger sind als unverschlüsselte.

Wenn man den kompletten verschlüsselten String innerhalb des Anwendungsprogramms speichern kann, braucht man sonst nichts, um die notwendigen Eingabefelder für eine erfolgreiche Entschlüsselung zu haben. Wenn Plattenplatz jedoch knapp ist, oder wenn das Dateiformat bei der Verschlüsselung nicht geändert werden kann, braucht man eine andere Technik. Auf den folgenden Seiten wird dargestellt, wie die negativen Effekte der Verlängerung des Geheimtextes im Rahmen einer ECB-Verschlüsselung reduziert werden können.

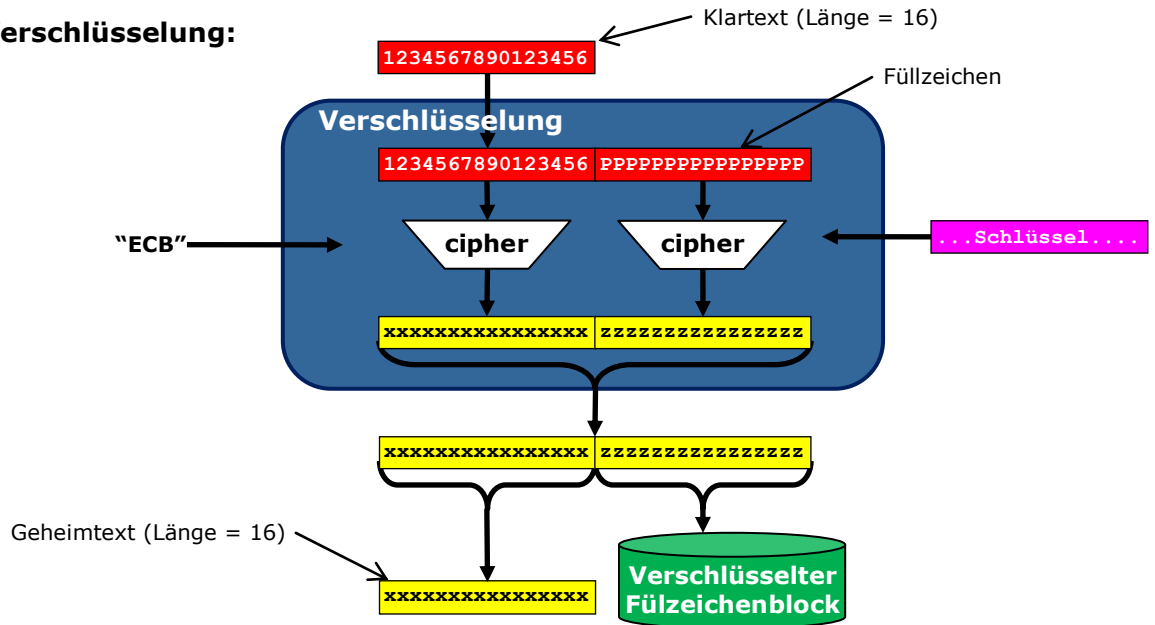
Die Entfernung des Füllzeichenblocks

Wenn der Klartext immer dieselbe Länge hat, und diese Länge zufällig genau ein Vielfaches von 16 beträgt, kann das Anwendungsprogramm die Verlängerung des Geheimtextes verhindern, indem eine einzelne Kopie der verschlüsselten Füllzeichen, aus denen der letzte Block des Geheimtextes besteht, gespeichert werden (siehe **Anhang D: Füllzeichen**).

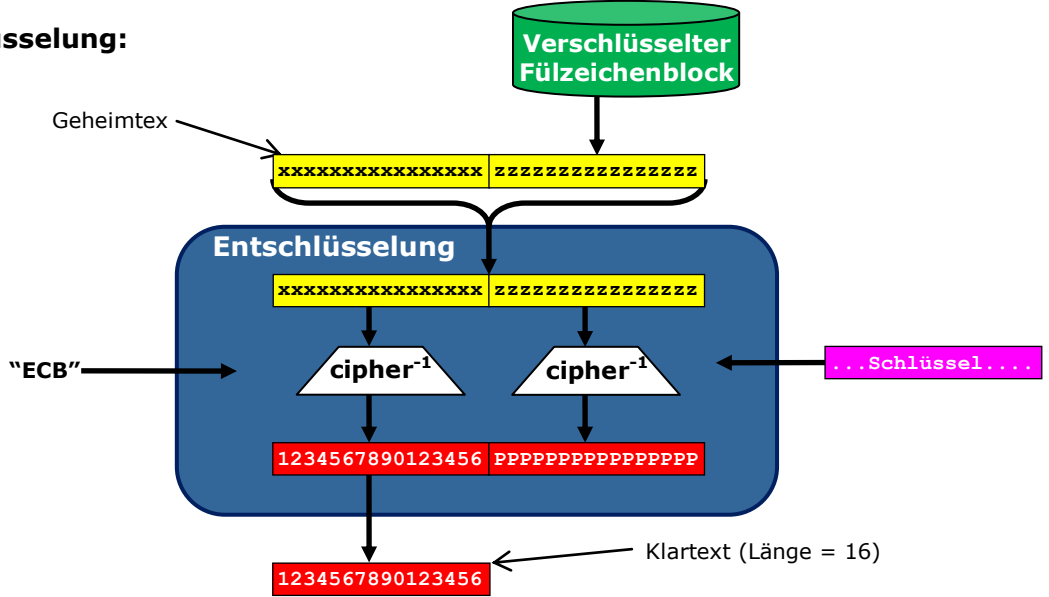
Da die AES-Verschlüsselung immer einen Gesamtblock von 16 Bytes auf einmal bearbeitet, indem von links nach rechts verarbeitet wird, hat dieser letzte Füllzeichenblock keinen Einfluss auf den Geheimtext, der sich links davon befindet, und wird für einen bestimmten Schlüssel immer gleich aussehen. Daher muss für jeden Schlüssel nur eine einzige Kopie des verschlüsselten Füllzeichenblocks gespeichert werden. All folgenden Verschlüsselungen von Klartext, dessen Länge ein Vielfaches von 16 beträgt, können für diesen Schlüssel die letzten 16 Zeichen des Geheimtext verwerfen, was bedeutet, dass der verbleibende Geheimtext dieselbe Länge wie der Klartext haben wird, bevor er verschlüsselt wurde.

Wenn ein Geheimtext entschlüsselt werden soll, der den letzten rechten Block nicht besitzt, wird einfach die gespeicherte Kopie des verschlüsselten Füllzeichenblocks rechts an den Geheimtext angehängt. Dann kann die Entschlüsselung unter Nutzung des umgekehrten Schlüssels wie gewohnt beendet werden. Hier ist das nochmal graphisch dargestellt:

ECB Verschlüsselung:



ECB Entschlüsselung:



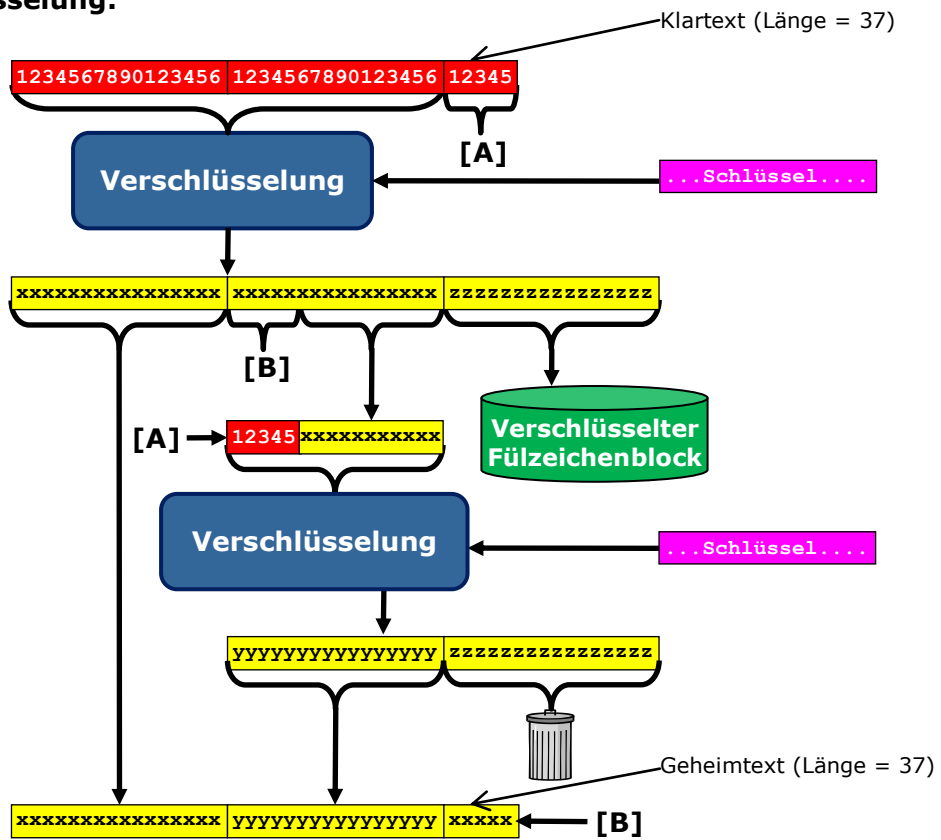
Gestohlener Geheimtext

Wenn man Klartext ECB-verschlüsseln will, dessen Länge nicht genau ein Vielfaches von 16 ist, aber jedenfalls größer als 16, kann eine Verlängerung des Geheimtext im Vergleich zum Klartext vermieden werden, wenn man eine Technik verwendet, die als [ciphertext stealing](#)^[12] bezeichnet wird.

Wenn man eine Verschlüsselung unter Verwendung von gestohlenem Geheimtext durchführen will, geht man wie folgt vor (siehe Abbildung unten):

1. Die Bytes am rechten Rand des Klartext, die nicht Teil eines vollständigen 16-Zeichen-Blocks sind, werden in den Hauptspeicher **[A]** übertragen.
2. Der gesamte Klartext wird ECB-verschlüsselt, außer den Zeichen, die in den Hauptspeicherbereich **[A]** übertragen wurden. Diese Verschlüsselung erzeugt einen weiteren 16-Byte langen, verschlüsselten Füllzeichenblock - siehe voriger Abschnitt **Die Entfernung des Füllzeichenblocks**.
3. Die Anzahl der Zeichen im Speicherbereich **[A]** wird ermittelt, und diese Anzahl Bytes wird vom Anfang des letzten 16-Byte-Geheimtext-Blocks in einen zweiten Speicherbereich **[B]** übertragen.
4. Die Geheimtext-Bytes, die in **[B]** (Schritt 3) gespeichert sind, werden mit den Klartext-Bytes aus **[A]** (Schritt 1) überschrieben.
5. Der letzte 16-Byte-Block des Geheimtext wird ECB-verschlüsselt (dieser Block enthält einige Bytes, die aus der vorhergehenden Verschlüsselung „gestohlen“ wurden). Es wird wiederum ein zusätzlicher, verschlüsselter Füllzeichenblock erzeugt. Dieser Block kann verworfen werden, da er mit dem in Schritt 2 erzeugten Block übereinstimmt.
6. Der ursprünglich letzte Geheimtext-Block wird durch den in Schritt 5 erzeugten Geheimtext-Block überschrieben.
7. Schließlich werden die Zeichen aus Speicherbereich **[B]** an das Ende des Geheimtextes angefügt.

ECB Verschlüsselung:



Klartext anderer Längen

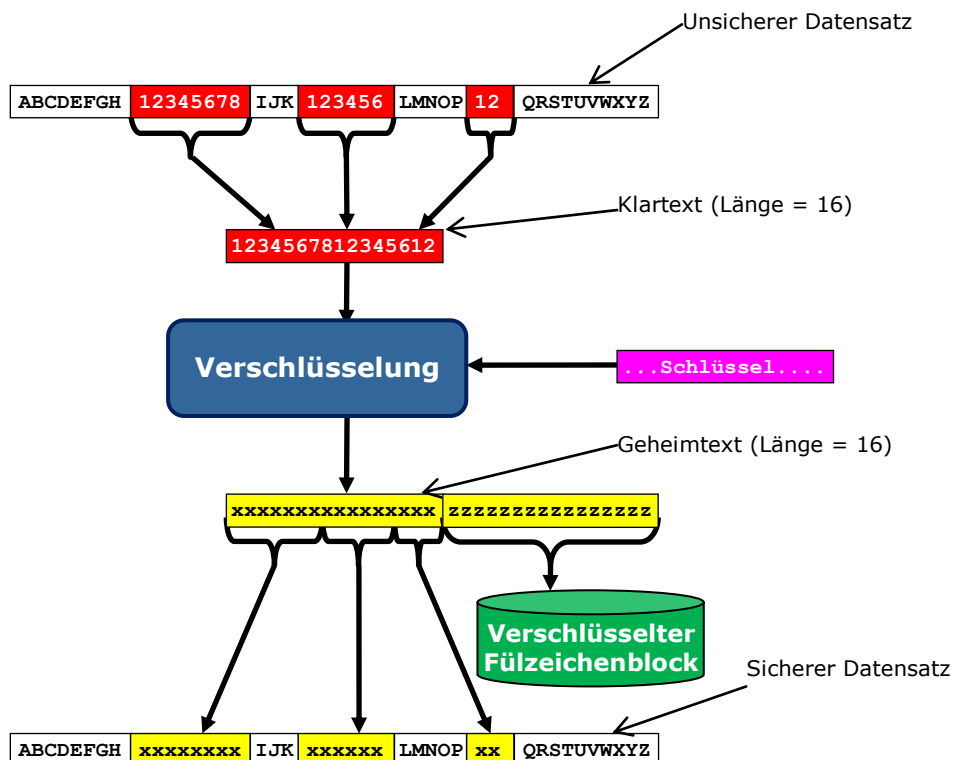
Für die ECB-Verschlüsselung und -Entschlüsselung von Klartextlängen von 1 bis 15 Bytes wird die Einbeziehung einiger zusätzlicher Felder im Anwendungsprogramm nötig sein, wenn eine Vergrößerung des Geheimtextes vermieden werden soll. Dies liegt daran, dass ein 16-Byte-Block die kleinstmögliche Eingabe für die AES-Verschlüsselung bzw. Entschlüsselung ist.

Blöcke mit Geheimtext müssen nicht als ein zusammenhängendes Feld innerhalb des Anwendungsprogramms gespeichert werden. Sie können aufgetrennt und in unterschiedlichen, kleineren Feldern gespeichert werden, um dann bei Bedarf für die Entschlüsselung wieder zusammengesetzt zu werden.

Daher besteht die einfachste Lösung für kurze Klartexte darin, etwas mehr Informationen als das absolute Minimum zu verschlüsseln. Wenn z.B. eine acht Byte lange Kontonummer verschlüsselt werden soll, warum nicht gleichzeitig auch die Bankleitzahl und/oder die Kontoart verschlüsseln. Sobald die kombinierte Länge einer Gruppe von Feldern mehr als 15 beträgt, kann ein kompletter Klartextblock von der Anwendung aufgebaut und als Eingabe in die Verschlüsselungsroutine verwendet werden.

Die Techniken zum Entfernen des Füllzeichenblocks und/oder zum Stehlen des Geheimtextes können verwendet werden, um einen Geheimtext von der gleichen Länge wie der Klartext zu erzeugen. Der Geheimtext kann dann über die Feldgruppe aufgelöst und anstelle des ursprünglichen Klartextes gespeichert werden. Siehe unten:

ECB Verschlüsselung:



Inkonsistente Geheimtexte

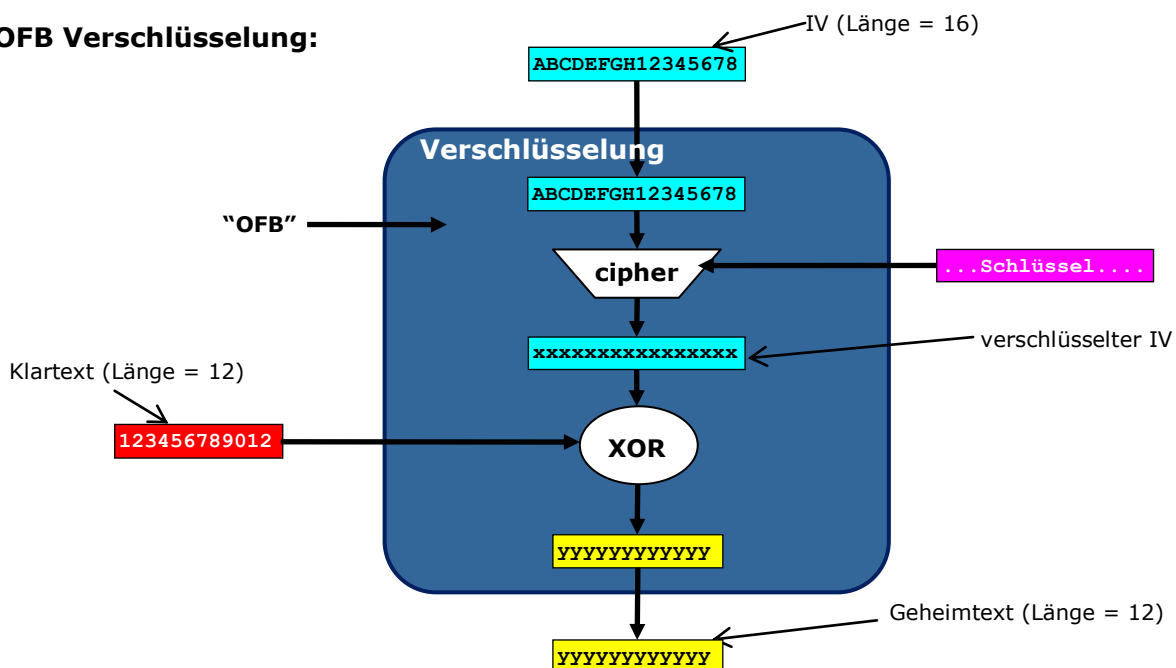
Die Erstellung eines inkonsistenten Geheimtextes aus demselben Klartext und Schlüssel beruht auf der Verwendung des AES Verschlüsselungsprozesses in Verbindung mit einem Operationsmodus, der nicht **Electronic Code Book (ECB)** Modus ist. (siehe **Anhang A: Vertraulichkeitsmodi**). Diese Vertraulichkeitsmodi benötigen einen **Initialisierungsvektor (IV)** (siehe **Anhang B: Initialisierungsvektoren**) oder **Zählerblock** (siehe **Anhang C: Zähler**), um ein den Verschlüsselungsprozess eine gewissen Unverhersagbarkeit einzuführen.

Die Nutzung des OFB-Modus

Von den verfügbaren inkonsistenten Vertraulichkeitsmodi wird **Output FeedBack (OFB)** häufig als der geeignetste für COBOL-Anwendungen angesehen. Dieser Modus erzeugt einen Geheimtext von gleicher Länge wie der Klartext (so dass es einfach ist, Klartextfelder durch den entsprechenden Geheimtext zu ersetzen), und der IV muss nicht unvorhersehbar sein - er muss nur einzigartig sein. Außerdem wird die effizientere Vorwärts-Chiffrierung sowohl innerhalb der Ver- als auch der Entschlüsselungs-Unterprogramme verwendet.

Um einen Klartext im OFB-Modus zu verschlüsseln, übergibt die Anwendung den IV, Modus ("OFB"), den zur Verschlüsselung zu verwendenden Schlüssel und den Klartext an das Verschlüsselungs-Unterprogramm. Das Unterprogramm gibt dann den entsprechenden, inkonsistenten, verschlüsselten Text zurück - siehe Beispiel unten:

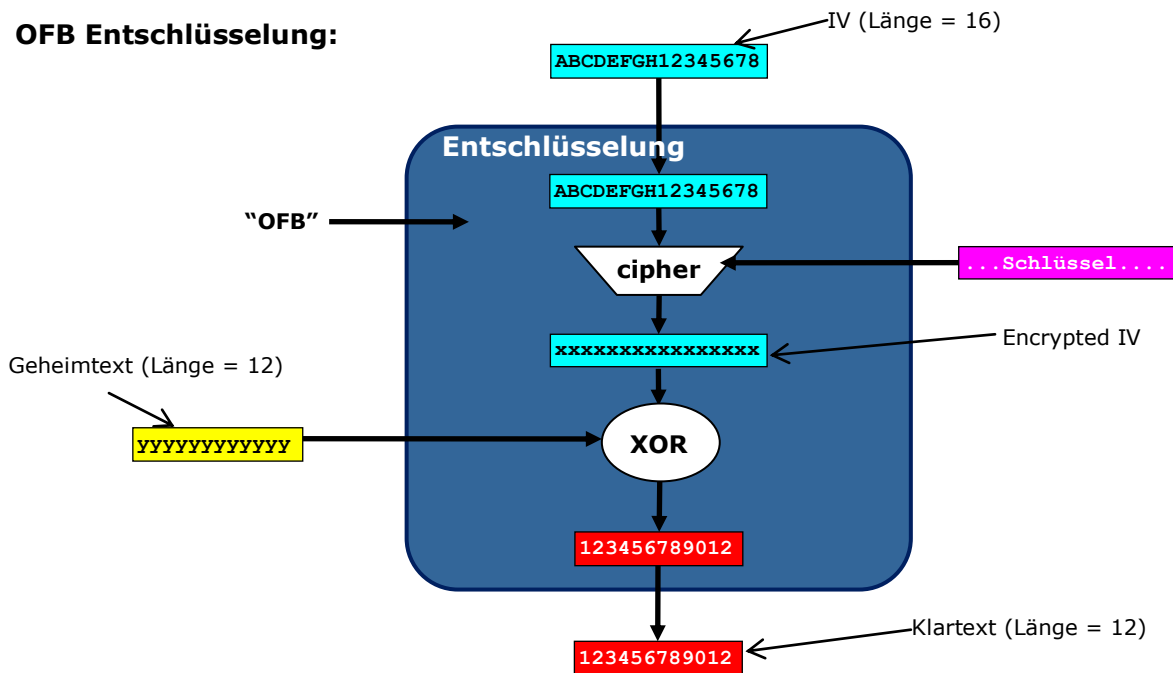
OFB Verschlüsselung:



Die obige Zeichnung verdeutlicht, wie wichtig der Initialisierungsvektor (IV) ist. Der IV, nicht der Klartext, wird von der AES Verschlüsselungsroutine bearbeitet. Um den Geheimtext zu erzeugen, wird jedes Klartextzeichen mit einem verschlüsselten IV-Zeichen kombiniert, mit der *Exklusiven Oder (XOR)* Logik (siehe **Anhang E: Exklusives Oder**). Sollte es im verschlüsselten IV noch unbenutzte Zeichen geben, werden diese nicht beachtet.

Zur Entschlüsselung werden derselbe IV, Modus und Schlüssel an das Entschlüsselungsprogramm übergeben, zusammen mit dem Geheimtext. Der Klartext wird dadurch zurückgegeben - siehe unten:

OFB Entschlüsselung:



Wie oben gezeigt, stellt die Entschlüsselungsroutine den verschlüsselten IV wieder her und kombiniert diesen dann mit dem Geheimtext unter Verwendung der *Exklusiven Oder* (**XOR**) Logik, die den Klartext wiederherstellt.

Die Kombination der Verwendung optimierter COBOL-Unterprogramme mit der AES-Vorwärts-Chiffre führt zu einer effizienten Ver- und Entschlüsselung. Da die AES-Verschlüsselungsroutine jedoch mit Blöcken von jeweils 16 Bytes (unabhängig von der Klartextlänge) arbeitet, kann durch die Kombination mehrerer kleiner Felder zu einem einzigen Klartextblock eine noch höhere Leistung erzielt werden - siehe Beispiele im vorherigen Abschnitt **Klartext anderer Längen** (Ignorieren von Füllzeichen). Mit einer einzigen Ausführung der Verschlüsselungsroutine können dann mehrere kleine Felder gleichzeitig ver- oder entschlüsselt werden.

Hinweis: Eine zusätzliche Überlegung beim Aufbau von inkonsistenten Geheimtexten ist, dass sie die Möglichkeit von „Kollisionen“ von Geheimtexten einführen. Als „Kollisionen“ von Geheimtexten bezeichnet man es, wenn zwei verschiedene Klartexte durch die Verschlüsselung den gleichen Geheimtext erzeugen. Dies wird für die meisten Anwendungen kein Problem darstellen, aber Systementwickler sollten sich dieser Möglichkeit bewusst sein, für den Fall, dass ein Geheimtext als Schlüsselfeld verwendet werden soll.

Verschlüsselung ohne Entschlüsselung

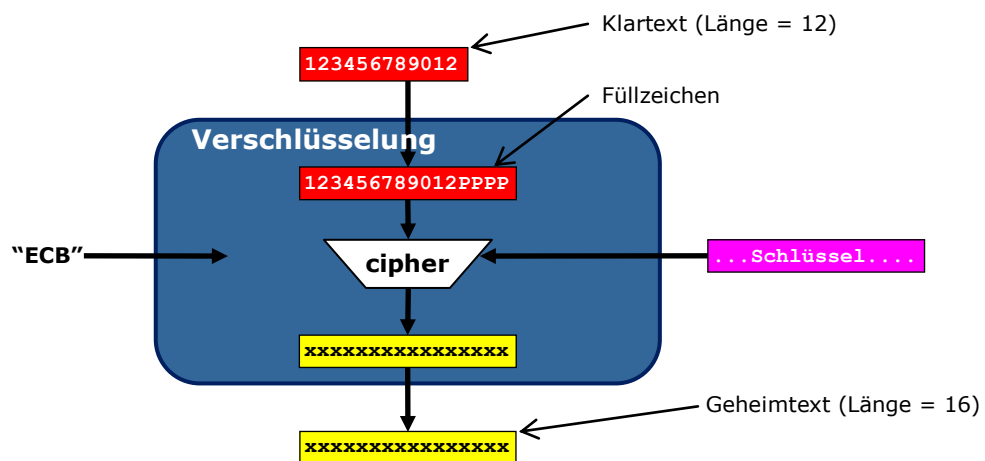
Wenn Eingaben wie etwa eine Nutzerkennung, ein Passwort oder eine PIN-Nummer authentifiziert werden müssen, ist es normalerweise nicht nötig, die Daten, die vom Computer gespeichert wurden, zu entschlüsseln. Man braucht nur eine Einweg-Verschlüsselung, oder eine Hashsumme der Eingabe, um einen Vergleich mit den gespeicherten Daten vornehmen zu können. Das führt zum Ergebnis „übereinstimmend“ oder „nicht übereinstimmend“.

Einwegverschlüsselung

Bei der Authentifizierung mit Einwegverschlüsselung ist es unabdingbar, dass für einen gegebenen Klartext immer derselbe Geheimtext generiert wird. Daher muss die AES-Verschlüsselungsroutine in Verbindung mit dem Betriebsmodus **Electronic Code Book (ECB)** verwendet werden (siehe **Anhang A: Vertraulichkeitsmodi**).

Um einen Klartext zu verschlüsseln, übergibt die Anwendung den Modus ("ECB"), den Schlüssel und den Klartext an das Verschlüsselungs-Unterprogramm. Das Unterprogramm gibt dann den entsprechenden konsistenten Geheimtext zurück - unten dargestellt:

ECB Verschlüsselung:



Ein Aspekt der ECB-Verschlüsselung ist, dass Füllzeichen (siehe **Anhang D: Füllzeichen**) immer rechts vom Klartext hinzugefügt werden, so dass vollständige Blöcke von 16 Bytes an die Verschlüsselungsroutine übergeben werden. Dadurch entsteht eine Geheimtextlänge, die auf das nächste Vielfache von 16 erweitert wird (auch wenn der Klartext bereits ein Vielfaches von 16 war).

Diese Vergrößerung des Geheimtextes kann ein Problem darstellen, wenn der Geheimtext anstelle des ursprünglichen Klartextes gespeichert werden soll. Der vollständige Geheimtext ist jedoch nur als Eingabewert für die Entschlüsselung notwendig. Wenn die Anwendung lediglich zweifelsfrei überprüfen muss, dass die richtige Nutzerkennung, das richtige Passwort oder die richtige PIN-Nummer eingegeben wurde, ist nur ein Vergleich mit einem Teil der Geheimtext-Bytes erforderlich.

Zum Beispiel würde eine 4-stellige PIN, die mit der ECB-Verschlüsselung bearbeitet wird, einen Geheimtext von 16 Bytes ergeben. Wenn von der Anwendung nur die Geheimtextbyte-Positionen 6 bis 9 gespeichert werden, ist das resultierende Bitmuster wahrscheinlich für jede der 10.000 verschiedenen PIN-Möglichkeiten eindeutig.

Hinweis: Eine Eingabeverifizierung unter Verwendung anderer Vertraulichkeitsmodi als ECB ist ebenfalls möglich, aber es wird der gleiche Initialisierungsvektor (IV) benötigt, um einen konsistenten Geheimtext zu erzeugen. Dies ist das einzige Szenario, in dem IV's jemals

wiederverwendet werden sollten. Wenn der erzeugte Geheimtext nicht mit dem gespeicherten Geheimtext übereinstimmt, muss der falsche Geheimtext vernichtet werden. Wenn ein neues Passwort oder eine neue PIN verschlüsselt werden soll, muss ein neuer, eindeutiger IV verwendet werden (siehe **Anhang B: Initialisierungsvektoren**).

Hash-Summe

Eine Alternative zur Verwendung von Einweg-AES-Verschlüsselung für die Eingabeauthentifizierung besteht darin, die Klartext-ID, das Passwort oder die PIN-Nummer mit einem der Standard-Hash-Summenalgorithmen zu leiten: SHA-1, SHA-224, SHA-256, SHA-384 oder SHA-512 (SHA-224, SHA-256, SHA-384 und SHA-512 sind zusammen als SHA-2 bekannt). Diese Algorithmen benötigen keinen Modus, Schlüssel oder Initialisierungsvektor, sie erzeugen lediglich eine konsistente binäre Hashsumme, die manchmal auch als „Message Digest“ bezeichnet wird und von den Bitwerten und Positionen innerhalb des Klartextstrings abhängt.

Die binäre Hash-Summe hat Ähnlichkeiten mit einem Geheimtext, kann aber nicht zur Wiederherstellung des ursprünglichen Klartextes verwendet werden, da der Hash-Algorithmus Informationen zerstört, die zur Umkehrung des Hash-Prozesses verwendet werden könnten.

Die Länge der Hash-Summe hängt von dem verwendeten Algorithmus ab. Mit Ausnahme von SHA-1, der eine Hash-Summe mit 160 Bit (20 Byte) Länge erzeugt, bezieht sich die Zahl im SHA-Namen auf die Anzahl der Bits in der erzeugten Hash-Summe. Daher erzeugt SHA-224 einen Hash von insgesamt 224 Bit (28 Bytes) Länge und SHA-512 einen Hash von 512 Bit (64 Bytes) Länge. Je länger der Hash, desto geringer ist die Wahrscheinlichkeit einer „Hash-Summenkollision“ (wenn zwei verschiedene Nachrichtentexte dieselbe Hash-Summe erzeugen).

Wie im Abschnitt **Einwegverschlüsselung** beschrieben, können einige der Zeichen an der äußersten rechten Position entfernt werden, wenn die Hash-Summe zu lang wird, um sie im Anwendungsprogramm einfach zu speichern. Das Anwendungsprogramm muss nur zweifelsfrei überprüfen, ob die korrekte Benutzerkennung, das richtige Passwort oder die richtige PIN-Nummer eingegeben wurde. Daher sollte eine binäre Hash-Summe, die auf die Länge des ID/Passwort/Pin-Feldes abgeschnitten wurde, ausreichen.

Hinweis: Hinweis: SHA-1, SHA-2 und SHA-3 Hash-Summen können mit dem [Redvers Hashing Algorithm](#)^[13] erzeugt werden.

Geheimtexte mit einem erhalten gebliebenem Format

Es gibt viele unterschiedliche Methoden, Geheimtext in ein spezielles Format zu konvertieren, und jede Methode hat die üblichen Vor- und Nachteile bezogen auf CPU-Nutzung, Speicherbedarf und Einzigartigkeit des Ergebnisses. Die Methoden, die sich für COBOL-Anwendungen eignen, gehören normalerweise zu einer der folgenden drei Gruppen: **Geheimtext-Sortierung**, **Radix-Aufteilung** und **Feistel Netzwerke**.

Geheimtext-Sortierung

Diese Methoden bestehen darin, eine einfache Nachschlagetabelle für jedes mögliche Klartext-Geheimtext-Paar zu erstellen und die Tabelle nach dem Geheimtext zu sortieren.

Vorteile:

- ✓ Beseitigt das Risiko von „Kollisionen“ im formatierten Geheimtext (wenn zwei verschiedene unformatierte Geheimtexte den gleichen Geheimtext erzeugen, wenn sie formatiert werden).
- ✓ Die Aufgaben der Ver- und Entschlüsselung werden zu einem einmaligen Batch-Ereignis verschoben, anstatt während der Lebensdauer des Schlüssels wiederholt aufgerufen zu werden.

Nachteile:

- ✗ Der ECB-Verschlüsselungsmodus muss verwendet werden, um eine konsistente Eins-zu-Eins-Beziehung zwischen jedem Klartext und dem zugehörigen Geheimtext herzustellen. Dies kann Auswirkungen auf die Sicherheit haben - siehe **Relevante Auswahlkriterien**.
- ✗ Der einmalige Batch-Prozess könnte hohe Computerressourcen erfordern, wenn die Bandbreite der möglichen formatierten Ergebnisse groß ist.
- ✗ Die endgültige Querverweistabelle muss sicher gespeichert werden.

Die Methode:

Diese Methode erfordert einen einmaligen Batch-Prozess wie folgt:

1. eine Tabelle oder Datei [**X**] erstellen, die jeden möglichen, gültigen, Klartext mit erhalten gebliebenem Format besteht (in der Regel in aufsteigender Reihenfolge, kann aber auch in beliebiger Reihenfolge sein).
2. für jeden Eintrag in der Tabelle/Datei [**X**] den Klartext bearbeiten unter Verwendung der AES-Verschlüsselung im ECB-Modus, und unter Verwendung desselben Schlüssels, und den resultierenden Geheimtext mit dem entsprechenden Klartext in einer neuen Tabelle/Datei [**Y**] speichern.
3. sortieren der Tabelle/Datei [**Y**], in Reihenfolge des Geheimtextes.
4. gleichzeitig alle Einträge in der Tabelle/Datei [**X**] und der Tabelle/Datei [**Y**] durchlesen, beginnend mit Eintrag 1, und eine dritte Tabelle/Datei [**Z**] aufbauen, die aus dem Klartext aus der Tabelle/Datei [**X**] und dem Klartext aus der Tabelle/Datei [**Y**] besteht.
5. Tabelle/Dateien [**X**] und [**Y**] verwerfen.

Die Anwendung verfügt nun über eine Querverweistabelle für die Ver- oder Entschlüsselung von Klartext oder formatiertem Geheimtext, indem der entsprechende Eintrag in der Tabelle/Datei [Z] nachgeschlagen wird.

Hinweis: Die Querverweistabelle [Z], die wie oben beschrieben erzeugt wurde, muss vollständig sicher vor unbefugtem Zugriff von außen aufbewahrt werden.

Beispiel 1:

Die Aufgabe besteht darin, die Codes der US-Bundesstaaten zu ver- und entschlüsseln, und dabei Codes zu erhalten, die gültige Codes der US-Bundesstaaten sind. Die drei Tabellen, die bei dieser Methode verwendet werden, würden wie unten abgebildet aussehen (Geheimtext ist hexadezimal dargestellt):

Tabelle: X	Tabelle: Y (vor der Sortierung)		Tabelle: Y (nach der Sortierung)		Tabelle: Z	
AK	AK	9ac644070198fc55 78f2b357323cbf62	WV	067ae50986c93774 8176b1a8a482650d	AK	WV
AL	AL	ff98c0d05e3dad3e b3d6236f23e74196	WI	3a361407219844ad 48f2750932ec81a3	AL	WI
AR	AR	47d2e1bf72264fa0 1fb274465e56ba20	AR	47d2e1bf72264fa0 1fb274465e56ba20	AR	AR
....
WI	WI	3a361407219844ad 48f2750932ec81a3	AK	9ac644070198fc55 78f2b357323cbf62	WI	AK
WV	WV	067ae50986c93774 8176b1a8a482650d	WY	b1a8a4820d052e1b 9a2614f23061320d	WV	WY
WY	WY	b1a8a4820d052e1b 9a2614f23061320d	AL	ff98c0d05e3dad3e b3d6236f23e74196	WY	AL

Hinweis: Es besteht durchaus die Wahrscheinlichkeit, dass ein oder mehrere Codes von Bundesstaaten an der gleichen Position in Tabelle X wie auch in der sortierten Tabelle Y erscheinen, was zu keiner Änderung durch die Ver- oder Entschlüsselung führt (siehe Code „AR“ oben). Dies spielt keine Rolle, solange kein Angreifer weiß, welche Codes sich geändert haben und welche nicht.

Beispiel 2:

Die Aufgabe (eine PCI-Anforderung) besteht darin, die Stellen 7 bis 12 von Kreditkartennummern zu ver- und entschlüsseln, um 6 verschlüsselte numerische Zeichen zu erhalten, ohne Kollisionen, so dass die verschlüsselten Zahlen anstelle der ursprünglichen Kreditkartennummern gespeichert werden können. Die drei Tabellen würden wie unten abgebildet aussehen (Geheimtext ist hexadezimal dargestellt):

Tabelle: X	Tabelle: Y (vor der Sortierung)		Tabelle: Y (nach der Sortierung)		Tabelle: Z
000000	000000	8d052e1b9a2614f2 b1a8a4823061320d	000001	0671a8a482ae5098 6c937748176b650d	000001
000001	000001	0671a8a482ae5098 6c937748176b650d	999998	3a340724ad48f275 0932e611984c81a3	999998
000002	000002	fad3eb3d623f98c0 d05e3d6f23e74196	999997	46ba27d2e1bf7226 01fb2744fa465e50	999997
....
999997	999997	46ba27d2e1bf7226 01fb2744fa465e50	000000	8d052e1b9a2614f2 b1a8a4823061320d	000000
999998	999998	3a340724ad48f275 0932e611984c81a3	999999	bf69a198fc5578f2 b35c6440707323c2	999999
999999	999999	bf69a198fc5578f2 b35c6440707323c2	000002	fad3eb3d623f98c0 d05e3d6f23e74196	000002

Hinweis: Im obigen Beispiel wurde der Speicherbedarf auf ein Minimum reduziert, indem lediglich der Wert aus Tabelle Y in Tabelle Z gespeichert wurde. Dies kann bei numerischen Klartexten geschehen, wenn die Tabelle X in Klartextreihenfolge aufgebaut ist. Der Wert aus Tabelle X ist lediglich die Positionsnummer, - 1. Die Verschlüsselung kann durch die Verwendung des 6-stelligen Klartextes, + 1, als Subskript zum direkten Nachschlagen des Geheimtextwertes erreicht werden. Bei der Entschlüsselung muss ein passender Wert in Tabelle Z gesucht und das resultierende Subskript, - 1, als Klartext verwendet werden. Wenn die Entschlüsselungseffizienz wichtig ist, kann eine umgekehrte Tabelle Z erstellt werden, indem die Werte der Tabelle X und der Tabelle Y in der Reihenfolge der Tabelle Y sortiert werden - dann werden die Werte der Tabelle Y verworfen.

Radix-Aufteilung

Diese Methode nutzt die Möglichkeiten in COBOL, Bit-Strings im Geheimtext als Ganzzahlen zu interpretieren, indem binäre (COMP) Felddefinitionen verwendet werden. Allgemein gesprochen, besteht die Vorgehensweise darin, die Ganzzahlen, die dem Geheimtext entsprechen, durch die Anzahl möglicher, gültiger Ergebnisse für die Reihenfolge jedes Zeichens zu dividieren (das wird Radix genannt), und den Restwert zu nutzen, um ein gültiges Zeichen für die Position nachzuschlagen.

Vorteile:

- ✓ Flexibel: Jede formatierte Zeichenposition wird unabhängig aufgebaut, wodurch eine Mischung verschiedener Zeichenbereiche je nach ihrer Position innerhalb des Feldes möglich ist.
- ✓ Minimale zusätzliche Verarbeitung.

Nachteile:

- ✗ Genau der erzeugte AES-Geheimtext muss ebenfalls innerhalb des Anwendungsprogramms gespeichert werden, um eine Entschlüsselung zu ermöglichen.
- ✗ Erhöht das Risiko von „Kollisionen“ des Geheimtextes (wenn zwei verschiedene unformatierte Geheimtexte bei der Formatierung denselben Geheimtext erzeugen) - dies kann für das Anwendungsprogramm von Bedeutung sein oder auch nicht.

Die Methode:

Diese Methode besteht aus dem folgenden ersten Schritt:

1. den Geheimtext in 8, 4, 2 oder 1-Byte-Zeichenketten zerlegen und diese Zeichenketten rechtsbündig in Felder verschieben, die mit LOW-VALUES Werten vorbelegt und als numerisches COMP ohne Vorzeichen redefined sind. Wie genau dies geschieht, hängt von der Länge des Geheimtextes und den verwendeten Kompilierungsoptionen ab. Das gewünschte Ergebnis ist eine Reihe von etwa gleich langen Binärfeldern, die das dezimale Äquivalent der Geheimtext-Bit-Einstellungen enthalten.

Dann, für jede formatierte Zeichenposition:

2. die Radix für die formatierte Zeichenposition bestimmen (ist möglicherweise nicht für alle Zeichenpositionen gleich). Z.B. hätte das Großbuchstaben-Alphabet ein Radix von 26.
3. eine Tabelle erstellen, die aus jedem gültigen Ausgabezeichen besteht. Z.B: „A“, „B“, „Z“.
4. das ganz linke COMP-Feld durch die Radix dividieren und den ursprünglichen Wert des COMP-Feldes durch den Quotienten ersetzen, um den Rest zu erhalten.
5. Zum Rest aus Schritt 4, 1 addieren, und dieses als Subskript verwenden, um das entsprechende gültige Zeichen aus der in Schritt 3 erstellten Tabelle nachzuschlagen. Platzieren Sie dieses Zeichen an die Stelle des formatierten Zeichens.

Die oben beschriebenen Schritte haben jedoch ein großes Problem: irgendwann wird das am weitesten links stehende COMP-Feld kleiner als die Radix - was dazu führt, dass jede nachfolgende Division zum gleichen Rest führt. Um das zu vermeiden, muss die Logik zum nächsten COMP-Feld nach rechts wechseln, sobald die Anzahl der Teilungen die Anzahl der Geheimtextzeichen erreicht, die sie repräsentieren. Z.B: Ein COMP-Feld, das 4 Geheimtextzeichen repräsentiert, sollte nie mehr als 4 Mal geteilt werden.

Hinweis: Bei der Entschlüsselung wäre eine Alternative zur Speicherung des ursprünglichen Geheimtextes im Anwendungsprogramm die Speicherung der restlichen COMP-Feldwerte. Der Geheimtext könnte dann wiederhergestellt werden, indem man die Methode in umgekehrter Richtung ausführt, mit der Radix multipliziert und den Rest addiert.

Beispiel:

Die Anforderung besteht darin, eine verschlüsselte 7-Byte-kanadische Postleitzahl in der Form „ANA NAN“ darzustellen, wobei „A“ ein beliebiger Groß- oder Kleinbuchstabe des Alphabets ist, „N“ eine numerische Ziffer von 0 bis 9 und das Leerzeichen ein Leerzeichen sein muss.

Der AES-Geheimtext ist hexadezimal: „e194afa36f35ca“. Dieser kann in die folgenden COMP-Felder unterteilt werden:

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

Tabelle für „A“:

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

Tabelle für „N“:

1	2	3	4	5	6	7	8	9	0
---	---	---	---	---	---	---	---	---	---

Tabelle für space:

space

Jede Zeichenposition wird dann wie folgt bearbeitet:

Position	Tabelle	Radix	COMP vorher	COMP danach	Rest	Subskript	Ausgabe
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
<i>Wechsle zum nächsten COMP-Feld</i>							
5	N	10	28469	2846	9	10	0
6	A	52	2846	54	38	39	m
<i>Wechsle zum nächsten COMP-Feld</i>							
7	N	10	202	20	2	3	3

Daher kann der Geheimtext „e194afa36f35ca“ als folgende Postleitzahl gespeichert werden: „P4j 0m3“.

Hinweis: Die oben beschriebene Methode und das Beispiel gehen davon aus, dass die Daten auf einer Big-Endian-Plattform gehalten werden. Bei Little-Endian-Plattformen müssen die binären Zeichenpositionen umgekehrt werden.

Feistel Netzwerke

Die Methoden des Feistel-Netzwerks kombinieren den Prozess der Verschlüsselung und des Formaterhalts, indem ein Teil des Klartextes wiederholt in Runden verschlüsselt wird und dann das Ergebnis durch „Modulus Addition“ dem unverschlüsselten Teil des Klartextes hinzugefügt wird. Die Entschlüsselung wird durch einen umgekehrten Lauf des Netzwerks erreicht, wobei das verschlüsselte Ergebnis „Modulus subtrahiert“ wird. Beide Prozesse werden in einer Reihe von „Runden“ durchgeführt.

Im Jahr 2013 gab das NIST die [Special Publication 800-38G^{\[15\]}](#) heraus, in der die Verwendung von Feistel-Netzwerken mit AES zur Herstellung sicherer formatkonservierender Geheimtexte formalisiert wurde. Es wurden drei spezifische Algorithmen definiert: FF1, FF2 und FF3; sie erfordern jeweils 10, 10 und 8 „Runden“. *Der FF2-Algorithmus wurde später aus der Spezifikation gestrichen.*

Die Fähigkeit, konsistente oder inkonsistente formatkonservierte Geheimtexte für einen gegebenen Klartext und Schlüssel zu erzeugen, wird durch die Verwendung eines „Tweaks“ gewährleistet. Tweaks sind ähnlich wie Initialisierungsvektoren (IV) - siehe **Anhang B: Initialisierungsvektoren**.

Konsistente/inkonsistente Geheimtexte sind besonders relevant für die formatkonservierte Verschlüsselung, da es bei der Erzeugung inkonsistenter Geheimtexte zu „Kollisionen“ kommen kann (wenn zwei verschiedene Klartexte denselben Geheimtext erzeugen). Die Entscheidung, ob konsistente oder inkonsistente Geheimtexte verwendet werden, sollte auf der Grundlage der Sicherheitsanforderungen der betreffenden Daten getroffen werden und nicht aufgrund der möglichen Unannehmlichkeiten von Geheimtext-Kollisionen - siehe Abschnitt **Relevante Auswahlkriterien**.

Vorteile:

- ✓ Geheimtexte bestehen aus Zeichen aus einem anwendungsdefinierten Alphabet.
- ✓ Die Länge der Geheimtexte ist gleich der Länge der Klartexte.
- ✓ Durch die Verwendung von konsistenten Geheimtexten können „Kollisionen“ vermieden werden.

Nachteile:

- ✗ Zusätzlicher Verarbeitungsbedarf durch wiederholte „Runden“ der Verschlüsselung.
- ✗ Geheimtext-„Kollisionen“ können unvermeidlich sein, wenn aus Sicherheitsgründen inkonsistente Geheimtexte erforderlich sind.
- ✗ Die Länge der Klartexte muss größer als ein Zeichen sein, was den Erhalt der Formatierung auf Zeichenebene verhindert.

Der letzte Punkt in der obigen Liste bezieht sich auf Felder wie Postleitzahlen oder Kalenderdaten, bei denen die Gültigkeit des Feldes nicht nur davon abhängt, dass es rein alphabetisch oder numerisch ist. Diese Schwierigkeit kann mit einem kleinen Anwendungscode vor und nach der Ver-/Entschlüsselung überwunden werden.

Beispiel 1:

Die Anforderung besteht darin, das Format einer 7-stelligen kanadischen Postleitzahl zu erhalten. Sie hat folgendes Format: „ANA NAN“, wobei „A“ ein beliebiger Großbuchstabe, „N“ eine Zahl und „ “ ein Leerzeichen ist.

Die Anwendung ordnet die Klartextzeichen in zwei jeweils drei Byte lange Zeichenketten um: „AAA“ und „NNN“. Die Verschlüsselung in Großbuchstaben unter Erhalt des Formats wird dann auf den String „AAA“ angewandt, und numerische Verschlüsselung unter Erhalt des Formats wird auf den String „NNN“ angewandt. Zum Schluss werden die verschlüsselten Zeichen wieder an ihre Ausgangspositionen zurückgestellt, wobei das Leerzeichen an Position 4 eingefügt wird.

Bei der Entschlüsselung würde der gleiche Prozess in umgekehrter Richtung ablaufen.

Beispiel 2:

Die Anforderung besteht darin, das Format eines 8-stelligen Geburtsdatums zu erhalten. Es hat das Format: „YYYYMMDD“ wobei „YYYY“ das Jahr ist, „MM“ ist die gültige Zahl eines Monats, und „DD“ ist ein gültiger Kalendertag innerhalb des Monats. Alle Geburtsdaten sind vor dem 1.1.2015.

Das Anwendungsprogramm führt die folgenden Schritte aus:

1. Konvertieren des Geburtsdatums in eine 5-stellige Ganzzahl, die der Anzahl der Tage seit 1900 entspricht, unter Verwendung einer COBOL FUNCTION oder einer Standard-Datumsroutine.
2. Die 5-stellige Ganzzahl wird in eine 6-stellige Zahl auf der Basis 6 umgewandelt, indem sie durch 6 geteilt wird, und zwar fünfmal, wobei jeder Rest an die nächste verfügbare Position ganz rechts gesetzt wird. Der letzte Quotient wird dann an die ganz linke Position gesetzt.
3. Die 6-stellige Zahl wird auf der Basis 6 verschlüsselt unter Verwendung einer numerischen, formaterhaltenden Verschlüsselung mit einem Zeichensatz von nur 0 bis 5.
4. Der 6-stellige Geheimtext wird wieder in eine ganze Zahl mit Basis 10 umgewandelt, indem die Summe berechnet wird: (7.776 x 1. Stelle, 1.296 x 2. Stelle, 216 x 3. Stelle, 36 x 4. Stelle, 6 x 5. Stelle, 1 x 6. Stelle).
5. Die 5-stellige Ganzzahl zur Basis 10 wird zurück in ein verschlüsseltes Geburtsdatum nach 1900 konvertiert, indem eine COBOL FUNCTION oder eine Standard-Datumsroutine des jeweiligen Ortes verwendet wird.
6. Wenn das Geburtsdatum aus Schritt 5 > 20150000 ist, wird der Vorgang ab Schritt 3 wiederholt, wobei der vorherigen Geheimtext ab Schritt 3 als neuen Klartext.

Schritt	Prozess	Wert
	Geburtsdatum:	19600101
1	Anzahl Tage seit 1900:	21914
2	Anzahl Tage in Basis 6:	245242
3	Anzahl Tage verschlüsselt:	543210
4	Verschlüsselte Tage in Basis 10:	44790
5	Verschlüsseltes Geburtsdatum:	20220819
6	Go to Step 3	
3	Anzahl Tage verschlüsselt:	012345
4	Verschlüsselte Tage in Basis 10:	1865
5	Verschlüsseltes Geburtsdatum:	19050209
6	Fertig	

Bei der Entschlüsselung läuft der gleiche Prozess in der gleichen Reihenfolge ab, einschließlich der Prüfung in Schritt 6; nur dass in Schritt 3 der eingegebene Geheimtext entschlüsselt wird.

Hinweis: Die oben beschriebene Methode konvertiert die Anzahl der Tage zu Basis 6 und verschlüsselt zu Basis 6, um die Wahrscheinlichkeit zu verringern, dass die Verschlüsselungen aus Schritt 3 wiederholt werden müssen (ca. 10%). Wenn die Effizienz kein Problem darstellt, können die Schritte 2 und 4 übersprungen werden und in Schritt 3 kann eine numerische Verschlüsselung zur Basis 10 erfolgen, so dass eine 60-prozentige Wahrscheinlichkeit besteht, alles ab Schritt 3 wiederholen zu müssen.

Beispiel 3:

Die Anforderung besteht darin, die Gültigkeitsanfangs- und Ablaufdaten von Kreditkarten als numerische Zeichen im „MMYY“-Format zu verschlüsseln, wobei kein Gültigkeitsanfangsdatum größer als das zugehörige Ablaufdatum ist.

Die Anwendung baut eine Tabelle auf, die aus Paaren gültiger Anfangs- und Ablaufdaten besteht. Wenn eine Liste länger als die andere ist, werden Nullen oder ein anderer ungültiger Wert den kürzeren Listeneinträgen hinzugefügt. Jedem Paar wird dann ein anderes alphanumerisches Zeichen zugewiesen - im Beispiel der 44 Einträge unten, „A“ bis „Z“, gefolgt von „a“ bis „r“:

ANFANGSDATUM	ABLAUFDATUM	alphanumerisches Zeichen
0112	0515	A
0212	0615	B
0312	0715	C
0412 bis 0315	0815 bis 0718	D bis Z & a bis m
0415	0818	n
0515	0918	o
0000	1018	p
0000	1118	q
0000	1218	r

Wenn ein Anfangs-/Ablaufdatum-Paar verschlüsselt werden muss, führt die Anwendung die folgenden Schritte aus:

1. Schlage das Anfangsdatum in der Tabelle unter der Spalte ANFANGSDATUM nach und setze das entsprechende alphanumerische Zeichen an die erste Klartextposition.
2. Schlage das Ablaufdatum in der Tabelle unter der Spalte ABLAUFDATUM nach und stelle das entsprechende alphanumerische Zeichen an die zweite Klartextposition.
3. Verschlüssele den zweistelligen Klartext formaterhaltend mit einem alphanumerischen Alphabet, das aus „A“ bis „Z“ und „a“ bis „r“ besteht.
4. Verwende das erste Zeichen des Geheimtextes, um ein verschlüsseltes Anfangsdatum in der Tabelle nachzuschlagen.
5. Verwende das zweite Zeichen des Geheimtextes, um ein verschlüsseltes Ablaufdatum in der Tabelle nachzuschlagen.
6. Wenn die Schritte 4 oder 5 zu einem ungültigen Datum (Nullen) führen, wiederhole ab Schritt 3 und verwende den vorherigen Geheimtext aus Schritt 3 als neuen Klartext.

Bei der Entschlüsselung wird der gleiche Prozess in der gleichen Reihenfolge und unter Verwendung der gleichen Tabelle einschließlich der Prüfung bei Schritt 6 ausgeführt, außer dass Schritt 3 den eingegebenen Geheimtext *entschlüsseln* würde.

Binäre Speicherbänke

Binäre Speicherbänke, die sich ideal für einen hohen Transaktionsdurchsatz eignen, bieten eine Alternative zur Echtzeit-Verschlüsselung mit der AES-Chiffre. Das Konzept beruht auf der Schaffung einer versteckten Reserve von pseudozufälligen Binärbytes, die in einer binären Operation „exclusive or“ mit vertraulichen Daten kombiniert werden. Das Ziel ist die Schaffung eines [One-Time-Pads](#)^[17], manchmal auch als [Vernam Cipher](#)^[18] bezeichnet.

Vorteile:

- ✓ Schnell und effizient.

Nachteile:

- ✗ Die Anwendung muss die verwendeten Byte-Zuweisungen innerhalb der Speicherbank verwalten.
- ✗ Möglichkeit von Geheimtext-"Kollisionen" (wenn zwei verschiedene Klartexte denselben Geheimtext erzeugen) - dies kann für die Anwendung wichtig sein oder auch nicht.
- ✗ Sicherer Speicherbereich ist für die Abspeicherung der Binärbank erforderlich.

Die Methode:

Dieser Ansatz erfordert die Erzeugung eines Arrays, Speicherbank genannt, von pseudozufälligen Binärdaten, die in einem gesicherten Bereich der Anwendung gehalten werden. Diese Speicherbank kann erstellt werden, indem ein anfänglicher, eindeutiger Klartext durch den AES-Verschlüsselungsprozess geleitet wird, der im **Counter**-Modus oder einem anderen Vertraulichkeitsmodus läuft (siehe [Anhang A: Vertraulichkeitsmodi](#)). Die Ausgabe wird nicht nur an den Anfang der binären Speicherbank kopiert, sondern auch als nächste Klartexteingabe für die AES-Verschlüsselung verwendet. Dieser Vorgang wird wiederholt, bis die Binäre Speicherbank als groß genug für alle vertraulichen Daten in der Anwendung angesehen wird. *Die Speicherbank kann bei Bedarf zu einem späteren Zeitpunkt erweitert werden, indem die letzte Ausgabezeichenfolge in der Speicherbank als nächste Klartexteingabe verwendet wird, um den Prozess der Binärspeicherbankgenerierung erneut zu starten.*

Wenn ein Anwendungsfeld verschlüsselt werden muss, wird seinem Inhalt ausschließlich eine entsprechende Anzahl von Bytes aus der binären Speicherbank zugewiesen. Diese Bytes werden dann mit *exklusiv or* verarbeitet (**XOR'd**) (siehe [Anhang E: Exklusives Oder](#)) gegen den Feldinhalt gesetzt, um den Geheimtext für das Feld zu erzeugen.

Die Entschlüsselung ist genauso einfach. Der Geheimtext wird *exklusiv or'd* (**XOR'd**) gegen die gleichen binären Speicherbank-Bytes, die im Verschlüsselungsverfahren zugewiesen wurden. Dies gibt den ursprünglichen Klartext zurück.

Die Implementierung einer binären Speicherbanklösung erfordert, dass die Anwendung die Zuweisungen von binären pseudo-zufälligen Speicherbank-Bytes zu den entsprechenden Anwendungsfeldern verwaltet und dabei gewährleistet, dass keine binären Speicherbank-Bytes mehr als einem Feld zugewiesen werden, es sei denn, es werden konsistente Geheimtexte benötigt - siehe [Relevante Auswahlkriterien](#). Die Anwendung muss auch die Verwendung der binären Speicherbank überwachen, indem sie Bytes entfernt, wenn sie nicht mehr benötigt werden, und die Speicherbank bei Bedarf erweitert.

Hinweis: Eine Alternative zur Verschlüsselung mit *exklusiv or*-Logik wäre die Addition des numerischen Wertes jeder Position eines binären Speicherbankzeichens zu jedem Klartextzeichen unter Verwendung eines Modulus, das dem Radix der Zeichenposition entspricht. Bei einem numerischen Klartextzeichen von „7“ und einem binären Speicherbankzeichenwert von 255 wäre das resultierende Geheimtextzeichen beispielsweise „2“ ($7 + 255 = 262$, Modulo 10 = 2). Diese Technik hat den zusätzlichen Vorteil, dass ein formatkonservierter Geheimtext erzeugt wird. Die Entschlüsselung erfolgt durch Modulo-Subtraktion des Wertes jeder binären Speicherbankzeichenposition.

Zusammenfassung

Wir hoffen, dass dieses Dokument die Eignung der in COBOL geschriebenen AES-Verschlüsselungssoftware zur Sicherung sensibler Daten gezeigt hat. Wir hatten das Ziel, mögliche Schwierigkeiten aufzuzeigen und Lösungen für diese Schwierigkeiten aufzuzeigen.

Um die wichtigsten Punkte zu rekapitulieren:

- In COBOL geschriebene Software-Verschlüsselung ist eine praktische, einfache und effiziente Option.
- AES ist der wichtigste, sichere globale Standard für die Verschlüsselung mit symmetrischem Schlüssel.
- Unter AES muss die Anwendung nur den Schlüssel geheim halten.
- Jedes AES-Verschlüsselungsprodukt sollte alle Vertraulichkeitsmodi und Schlüssellängen unterstützen.
- Wenn Füllzeichen erforderlich sind, sollten sie den [Public-Key Cryptography Standards \(PKCS#7\)](#)^[19] entsprechen.
- Wenn Geheimtext in numerischen oder alphanumerischen Feldern übertragen oder gespeichert werden soll, muss er formatiert oder in hexadezimale oder Base64-Felder umgewandelt werden.
- Speicherbereiche, die entschlüsselte Klartexte oder Schlüssel enthalten, sollten nach Abschluss des Ver-/Entschlüsselungsvorgangs gelöscht werden.

Glücklicherweise ist das [Redvers Encryption Module](#)^[20] ein [NIST validated](#)^[21], AES-Softwareprodukt, in COBOL geschrieben und für COBOL-Anwendungen konzipiert. Es unterstützt die primären Vertraulichkeitsmodi (ECB, CBC, CFB, OFB und CTR), Festformat-Verschlüsselung und MAC/CCM-Modi für verschlüsselungsbasierte Hash und Authentifizierung. Alle Schlüssellängen (128, 192 oder 256 Bit) und Pad-Zeichen-Methoden werden ebenfalls unterstützt, und es enthält eine „reinige den Speicher“-Funktion zum Schutz von Klartexten und Schlüsseln.

Sie können eine kostenlose, unverbindliche 30-Tage-Testversion des Redvers Encryption Module herunterladen: https://www.cobol.de/data_encryption_free_trial.php

Wenn Sie Fragen an uns haben, benutzen Sie bitte unsere Website Kontakt: <https://www.cobol.de/contact.php>

Anhang A: Vertraulichkeitsmodi

Die NIST [Special Publication 800-38A](#)^[22] enthält eine genaue Beschreibung jedes der fünf AES-Vertraulichkeitsmodi. Die Entscheidung, welcher Modus zu verwenden ist, hängt von einer Kombination aus Unternehmenspolitik, externen Anbietern und Systemanforderungen ab. Um diese Entscheidung zu erleichtern, wird in den folgenden Abschnitten hervorgehoben, wie die verschiedenen Vertraulichkeitsmodi innerhalb einer COBOL-Anwendungsumgebung am besten genutzt werden können.

Electronic Code Book (ECB)

Abgesehen davon, dass dieser Modus keinen Initialisierungsvektor oder Zähler benötigt, ist der wichtigste Aspekt des **Electronic Code Book** - Modus, dass er für einen gegebenen Klartext und Schlüssel immer den gleichen Geheimtext erzeugt. Auch wenn das bedeutet, dass ECB nicht unbedingt der sicherste verfügbare Modus ist, ist ein konsistenter Geheimtext für den Abgleich mit anderen verschlüsselten Werten innerhalb der Anwendung unerlässlich.

Ein weiterer Aspekt des **Electronic Code Book** - Modus ist, dass es der einzige Vertraulichkeitsmodus ist, der für einen gegebenen Klartext und Schlüssel einen eindeutigen Geheimtext garantiert, was die Möglichkeit von Geheimtext-"Kollisionen" ausschließt (wenn zwei verschiedene Klartexte bei der Verschlüsselung den gleichen Geheimtext erzeugen). Dies kann für das Anwendungsprogramm möglicherweise von Bedeutung sein.

Zur Entschlüsselung von Geheimtext, der im **Electronic Code Book** - Modus erstellt wurde, muss die inverse AES-Chiffrierung verwendet werden. Diese inverse Chiffrierung ist wegen der komplexeren binären Arithmetik mit einem etwas höheren Rechenaufwand verbunden als die Vorwärts-Chiffrierung. Die Modi **Cipher Feedback**, **Output Feedback** und **Counter** vermeiden diesen Rechenaufwand, indem die Vorwärts-Chiffrierung für die Ver- und Entschlüsselung verwendet wird.

Ein weiterer potenzieller Nachteil des **Electronic Code Book** - Modus ist, dass er den Klartext und den Geheimtext in vollständigen Blöcken von 16 Bytes an die AES-Chiffrierung oder inverse Chiffrierung übergibt. Dies erfordert die Einfügung von 1 bis 16 Füllzeichen (siehe **Anhang D: Füllzeichen**) am rechten Ende des Klartextes, wenn dieser Modus verwendet wird (auch wenn der Klartext ein Vielfaches von 16 Bytes lang ist). Dieser erweiterte Klartext erzeugt daher einen erweiterten Geheimtext, der immer länger als der ursprüngliche Klartext sein wird - hier sei auf die Abschnitte innerhalb von **Konsistente Geheimtexte** verwiesen, wenn dies ein Problem darstellt.

Die Art und Weise, in der der **Electronic Code Book** - Modus Klartext oder Geheimtext in ganzen Blöcken von 16 Bytes unabhängig voneinander verarbeitet, bedeutet, dass Blöcke gleichzeitig ver- und entschlüsselt werden können, wobei also eine parallele Verarbeitung erfolgt. Der potenzielle Nachteil ist, dass Blöcke von 16 Bytes von einem Angreifer aus dem Geheimtext entfernt werden könnten, ohne den legitimen Entschlüsselungsprozess zu stören. Die anderen Vertraulichkeitsmodi weisen diese Schwäche nicht auf, da sie Blöcke von einer Verschlüsselungsstufe zur nächsten verketteten.

Cipher Block Chaining (CBC)

Cipher Block Chaining erfordert einen zufällig erzeugten Initialisierungsvektor (IV) (siehe **Anhang B: Initialisierungsvektoren**) für die Ver- und Entschlüsselung. Die Verwendung des IV erzeugt für jeden Klartext einen anderen Geheimtext.

Wie beim **Electronic Code Book** wird beim **Cipher Block Chaining** die inverse AES-Verschlüsselung zur Entschlüsselung des Geheimtextes verwendet. Dies bedeutet einen etwas höheren Rechenaufwand als bei der Verwendung der Vorwärts-Chiffre zur Entschlüsselung.

Ebenfalls in Übereinstimmung mit dem **Electronic Code Book** werden beim **Cipher Block Chaining** Klartext und Geheimtext in vollständigen Blöcken von 16 Bytes durch die AES-Chiffre oder inverse Chiffre geleitet. Dies erfordert die Hinzufügung von Füllzeichen (siehe **Anhang D: Füllzeichen**) und erzeugt einen erweiterten Geheimtext - siehe die Abschnitte innerhalb des Kapitels **Konsistente Geheimtexte**, wenn dies zu Schwierigkeiten führen sollte.

Cipher Feedback (CFB)

Vom **Cipher Feedback** Modus gibt es vier unterschiedliche Varianten: 1-Bit-Modus, 8-Bit-Modus, 64-Bit-Modus oder 128-Bit-Modus. Die Bitanzahl bezieht sich auf die Anzahl verschlüsselter Bits, die von jedem 128-Bit-AES-Chiffrierungsausgangsblock verwendet werden. Alle übrigen Bits werden verworfen. Daher gilt bei einer Klartextlänge von 16 Byte (128 Bit):

- Der 1-Bit-Modus erfordert 128 AES-Chiffre-Ausführungen zum Ver-/Entschlüsseln.
- Im 8-Bit-Modus sind 16 AES-Chiffre-Ausführungen zum Ver-/Entschlüsseln erforderlich.
- Im 64-Bit-Modus sind 2 AES-Chiffre-Ausführungen zum Ver-/Entschlüsseln erforderlich.
- Im 128-Bit-Modus ist für die Ver-/Entschlüsselung 1 AES-Chiffre-Ausführung erforderlich.

Praktische Anwendungen für den 1-Bit-Modus sind aufgrund des enormen Verarbeitungsaufwands selten, aber wenn nur gelegentlich eine Verschlüsselung erforderlich ist oder für die Verschlüsselung bestimmter Bitpositionen in einer Klartextzeichenfolge, könnte dieser Modus gewählt werden.

Im 8-Bit-Modus wird die AES-Chiffre auf Zeichenebene ausgeführt, wodurch die Notwendigkeit von Füllzeichen (siehe **Anhang D: Füllzeichen**) und die entsprechende Erweiterung des Geheimtextes vermieden wird. Der Overhead bei der Ausführung der Chiffrierung einmal für jedes Klartextzeichen ist der einzige grundlegende Nachteil.

Die 64-Bit- und 128-Bit-Modi sind natürlich schneller als die 1-Bit- und 8-Bit-Modi, aber diese Modi erfordern die Hinzufügung von Füllzeichen und erfordern daher eine Geheimtexterweiterung. Im 64-Bit-Modus müssen jedoch nur maximal 8 Füllzeichen hinzugefügt werden, um einen Klartext-Halbblock zu vervollständigen.

Wie beim **Cipher Block Chaining** erfordert der **Cipher Feedback Modus** einen zufällig erzeugten Initialisierungsvektor (IV) (siehe **Anhang B: Initialisierungsvektoren**) für die Ver- und Entschlüsselung. Durch die Verwendung des IV wird für jeden Klartext ein anderer Geheimtext erzeugt.

Im Gegensatz zum **Cipher Block Chaining** verwendet der **Cipher Feedback** - Modus die effizientere AES-Vorwärts-Chiffrierung zur Ver- und Entschlüsselung von Klartexten.

Hinweis: Obwohl in der NIST [Special Publication 800-38A](#)^[22] Anhang A steht: „der Klartext muss eine Folge von einem oder mehreren vollständigen Datenblöcken (oder, für den CFB-Modus, Datensegmenten) sein“, benötigt der 64-Bit- und 128-Bit-CFB-Modus eigentlich kein vollständiges endgültiges Datensegment, um zu funktionieren. Die endgültige Operation bei der CFB-Verschlüsselung und -Entschlüsselung ist ein *exklusives oder* (siehe **Anhang E: Exklusives oder**) zwischen der AES-Chiffre-Ausgabe und dem Klartext oder Geheimtext. Die Hinzufügung einer einfachen Logik zum AES-Chiffre könnte das *Exklusive oder* auf die Anzahl der Klartext- oder Geheimtextzeichen im endgültigen Segment beschränken, ohne dass Füllzeichen erforderlich sind.

Output Feedback (OFB)

Der **Output Feedback** Modus erfordert einen eindeutigen Initialisierungsvektor, um für jeden Klartext einen anderen Geheimtext zu erzeugen (siehe **Anhang B: Initialisierungsvektoren**). Dieser Modus erfordert keine Hinzufügung von Füllzeichen und erzeugt so einen Geheimtext, der die gleiche Länge wie der Klartext hat. Er verwendet auch die effizientere Vorwärts-Chiffrierung sowohl für die Ver- als auch für die Entschlüsselung.

Dieser Modus wird häufig für COBOL-Anwendungen gewählt und wird im Abschnitt **Verwendung des OFB-Modus** näher erläutert.

Counter (CTR)

Der Counter (Zähler)-Modus bietet COBOL-Anwendungen eine gute Alternative zum **Output Feedback** Modus. Er erfordert einen Zählerblock (siehe **Anhang C: Zähler**) anstelle eines Initialisierungsvektors, um für jeden Klartext einen anderen Geheimtext zu erzeugen. Wie der **Output Feedback** Modus erfordert dieser Modus keine Hinzufügung von Füllzeichen und verwendet die effizientere Vorwärts-Chiffrierung sowohl für die Ver- als auch für die Entschlüsselung.

Anhang B: Initialisierungsvektoren

Initialisierungsvektoren (IVs) wurden nicht erfunden, nur um den Anwendungsentwicklern das Leben schwer zu machen. Sie verbessern den Verschlüsselungsprozess, indem sie Anwendungen in die Lage versetzen, inkonsistente Geheimtexte zu erzeugen, und sie sind für die Erstellung von Geheimtexten, die gleich lang sind wie die entsprechenden Klartexte, von entscheidender Bedeutung - besonders wichtig für COBOL-Anwendungen, deren Feldformate eine feste Länge haben.

Was sind IVs?

- IV's sind ein Datenblock von 16 Byte Länge, der für die Ver- und Entschlüsselung im CBC-, CFB- und OFB-Modus benötigt wird.
- Für jede Ver- oder Entschlüsselung ist nur ein IV erforderlich, unabhängig von der Länge des Klar- bzw. Geheimtextes.
- Ein IV hat kein festes Format. Er kann aus alphanumerischen und/oder binären Zeichen in beliebiger Reihenfolge bestehen.
- Der zur Entschlüsselung verwendete IV muss der gleiche sein wie der zur Verschlüsselung verwendete.
- Ein IV muss nicht von der Außenwelt geheim gehalten werden.
- Jeder IV muss für jede Verschlüsselungsoperation, für einen bestimmten Schlüssel, einmalig (eine Nonce) sein. *Dies muss für alle Anwendungen gelten, die denselben Schlüssel verwenden.*
- IV's, die für die CBC- und CFB-Modus-Verschlüsselung verwendet werden, dürfen nicht vorhersehbar sein, d.h. ein potentieller Eindringling darf nicht in der Lage sein, mehrere potentielle IV's als Teil eines IV's, die für die **CBC-** und **CFB-**Modus-Verschlüsselung verwendet werden, dürfen nicht vorhersehbar sein, d.h. ein potentieller Eindringling darf nicht in der Lage sein, mehrere potentielle IV's als Teil eines [Brute-force attack](#)^[11].

Die Erzeugung eines eindeutigen IV stellt für COBOL-Anwendungen in der Regel kein Problem dar. Die für die Verschlüsselung ausgewählten Felder werden in der Regel mit genügend Schlüsselinformationen verbunden, um ihre Position innerhalb der Anwendungsdatenbank oder des Dateiindexes eindeutig zu identifizieren. Die Verkettung dieser Schlüsselfelder ergibt eine Zeichenfolge, die den Speicherort des Feldes eindeutig bestimmt. Dann wird dem eine Datums/Zeit- oder Folgennummer für die Verschlüsselung hinzugefügt, und alles, was gebraucht wird, um einen sicheren, eindeutigen IV zu erstellen, ist vorhanden.

Die Felder, aus denen sich der IV zusammensetzt, können in beliebiger Reihenfolge und in beliebigem Format sein. Wenn also dieselben Schlüsseldetails für mehr als eine Verschlüsselung verwendet werden sollen, ist eine Neuordnung oder Neuformatierung der betreffenden Felder ausreichend, um einen weiteren eindeutigen IV für zusätzliche Verschlüsselungen zu erzeugen. Die entschlüsselnde Anwendung muss jedoch dieselbe Feldreihenfolge und dieselben Formate verwenden, um den richtigen IV für eine erfolgreiche Entschlüsselung zu erhalten.

Wenn die Gesamtlänge aller Felder, die zur Erzeugung eines eindeutigen IV benötigt werden, mehr als 16 Bytes beträgt, können unterschiedliche Methoden verwendet werden, um den IV-String zu verkürzen:

- Komprimieren von numerischen Daten, indem sie in Felder verschoben werden, die als gepackt (COMP-3) oder binär (COMP) definiert sind.
- Multiplizieren von mehrfach gewichteten numerischen Werten, um einen einzelnen, größeren Produktwert zu erzeugen.

- Generieren von eindeutigen Hash-Summen, um alphanumerische Zeichenfolgen darzustellen.

Wenn es nicht möglich ist, einen eindeutigen IV aus unverschlüsselten Schlüsselfeldern zu erstellen, oder wenn die entschlüsselnde Anwendung keinen Zugang zu den Feldern und Informationen hat, aus denen sich der IV zusammensetzt, kann ein IV anderweitig erstellt und unverschlüsselt über einen anderen Kommunikationsweg oder am Anfang des Geheimtextes an die zur Entschlüsselung benutzte Anwendung gesendet werden (IV's müssen nicht geheim gehalten werden).

Warum müssen IV's einzigartig sein?

Es gibt mehrere Gründe, warum jeder IV für jede OFB-Verschlüsselung einzigartig sein muss. Das hängt damit zusammen, dass das, was innerhalb der Verschlüsselungsroutine durch die Verschlüsselungs-Chiffre-Logik verarbeitet wird, der IV und nicht etwa der Klartext ist.

Wie in der **Zusammenfassung** erwähnt wurde, handelt es sich bei der AES-Verschlüsselung um einen Prozess, bei dem Daten, die geheim gehalten werden müssen, mit einer pseudo-zufälligen Bitfolge kombiniert werden. Im OFB-Modus ist es der verschlüsselte IV, die diese pseudozufällige Zeichenfolge liefert, die dann mit dem Klartext unter Verwendung der *Exklusiv-oder* (XOR)-Logik (siehe **Anhang E: Exklusives Oder**) kombiniert wird, um den Geheimtext zu erzeugen - siehe die Darstellung im Abschnitt **Verwendung des OFB-Modus**. Wenn also derselbe IV als Eingabe für die Verschlüsselungsroutine dient, wobei derselbe Schlüssel verwendet wird, ist der verschlüsselte IV auch derselbe.

Betrachten wir nun die Folgen der Verwendung von doppelten IVs genauer: Die Gesetze der Binärmathematik bedeuten, dass, wenn zwei verschiedene, allgemein bekannte Geheimtexte, die aus demselben IV erzeugt wurden, zusammen XOR'd würden, das Ergebnis eine binäre Karte des genauen Unterschieds zwischen den beiden ursprünglichen Klartexten wäre - wobei die Wirkung des IV, der Chiffre und des Schlüssels aufgehoben würde! Von diesem Punkt aus wäre es nicht schwierig, den gesamten vertraulichen Klartext abzuleiten, der an die Verschlüsselungsprozesse weitergegeben wurde, die denselben IV verwendet haben.

Um den oben angeführten Punkt zu beweisen, können wir uns ansehen, was passiert, wenn zwei einzelne Zeichen, „A“ und „Z“, mit derselben IV (vereinfacht auf ein Zeichen) verschlüsselt werden. Nehmen wir zunächst an, dass der duplizierte IV, die durch die Verschlüsselungschiffre geleitet wurde, eine verschlüsselte IV-Binärzeichenfolge von „10101010“ ergibt. Unter Verwendung der EBCDIC-Darstellung ist der Buchstabe „A“ binär „11000001“ und der Buchstabe „Z“ binär „11101001“:

Am Ende des Verschlüsselungsprozesses wird der verschlüsselte IV mit der Klartext-Binärdatei „A“ XOR'd, was Geheimtext A ergibt:

```
Verschlüsselter IV: 1 0 1 0 1 0 1 0
Klartext-A:         1 1 0 0 0 0 0 1
                   -----
Geheimtext-A:     0 1 1 0 1 0 1 1
                   -----
```

In einem anderen Verschlüsselungsverfahren wird derselbe verschlüsselte IV mit dem Klartext-Binärcode „Z“ XOR'd, was Geheimtext-Z ergibt:

```
Verschlüsselter IV: 1 0 1 0 1 0 1 0
Klartext-Z:         1 1 1 0 1 0 0 1
                   -----
Geheimtext-Z:     0 1 0 0 0 0 1 1
                   -----
```

Nun wenden wir XOR auf den öffentlich bekannter Geheimtext-A mit dem öffentlich bekannten Geheimtext-Z an:

```
Geheimtext-A:    0 1 1 0 1 0 1 1
Geheimtext-Z:    0 1 0 0 0 0 1 1
-----
XOR Ergebnis:    0 0 1 0 1 0 0 0
-----
```

Das Ergebnis zeigt die Klartext-Bitpositionen (3. und 5. von links) an, die geändert werden müssen, um ein „A“ im Klartext in ein „Z“ im Klartext oder ein „Z“ im Klartext in ein „A“ im Klartext zu verwandeln.

Anhang C: Zähler

Zählerblöcke werden bei der Ver- und Entschlüsselung im **Counter (CTR)-Modus** anstelle eines Initialisierungsvektor (IV)-Parameters verwendet (siehe **Anhang B: Initialisierungsvektoren**). Wie IVs müssen Zähler ein eindeutiger 16-Byte-Block sein. Sie können in jedem beliebigen Format vorliegen und müssen nicht geheim gehalten werden. Der gleiche Zählerblock, der für die Verschlüsselung verwendet wird, ist auch für die Entschlüsselung erforderlich.

Im Gegensatz zu den IVs ist für jeweils 16 Byte Eingabedaten (Klartext oder Geheimtext) ein unterschiedlicher Zählerblock erforderlich. Gute Verschlüsselungssoftware wie das [Redvers Encryption Module](#)^[20] erhöht den ersten Zählerblock, der an die Unterroutine übergeben wird, intern um eine binäre 1 für jeweils 16 weitere Bytes der Eingabe. Der letzte Zählerblock + Binär 1 wird dann an die Anwendung zurückgegeben, um von der nächsten Verschlüsselung im Counter-Modus verwendet zu werden.

Ein weiterer Unterschied zwischen der Verwendung eines Zählers und IV besteht darin, dass zur Gewährleistung der Eindeutigkeit von Zählerblöcken deren Verwendung zentral verwaltet werden muss, so dass jede Verschlüsselung einen anderen Bereich von Zählerwerten verwendet. Dies kann einzelne Streaming-Anwendungen erfordern, die den gleichen Verschlüsselungsschlüssel verwenden.

Da der erste Zählerblock, der für die Verschlüsselung verwendet wird, auch für die Entschlüsselung benötigt wird, muss er an die entschlüsselnde Anwendung übergeben werden. Dies kann entweder erfolgen, indem er an den Anfang des Geheimtextes gesetzt wird, oder indem ein anderer Kommunikationsweg benutzt wird. Stattdessen können sowohl innerhalb der Verschlüsselungs- als auch der Entschlüsselungsanwendung synchronisierte Zähler beibehalten werden.

Ein typischer Einsatz der Counter-Modus-Verschlüsselung ist die Erzeugung eines Vorrats pseudozufälliger binärer Strings, die zu Zeiten hoher Belastung bereit stehen, Klartext mit *exklusivem oder* (siehe **Anhang E: Exklusives Oder**) zu bearbeiten, um eine schnelle Ver- und Entschlüsselung zu gewährleisten - siehe **Binäre Speicherbänke**.

Anhang D: Füllzeichen

Zur Verschlüsselung mit den Vertraulichkeitsmodi **Electronic Code Book (ECB)**, **Cipher Block Chaining (CBC)** und manchmal **Cipher Feedback (CFB)** (siehe [Anhang A: Vertraulichkeitsmodi](#)) werden rechts vom ursprünglichen Klartext Füllzeichen hinzugefügt, damit dieser ausschließlich aus 16-Byte-Blöcken besteht. Das nennt sich kurz „Padding“. Diese zusätzlichen Füllzeichen werden am Ende des Entschlüsselungsverfahrens entfernt, so dass die Klartextlänge wieder hergestellt wird.

Wenn die Klartextlänge bereits genau ein Vielfaches von 16 ist, wird ein weiterer Block von 16 Füllzeichen hinzugefügt, damit der Entschlüsselungsprozess das Ende des Klartextes richtig identifizieren kann. Ohne die Gewissheit, dass die Ausgabe der Entschlüsselungschiffre mit einem Füllzeichen endet, wäre es unmöglich, ein Füllzeichen vom letzten Datenbyte zu unterscheiden.

Das Hinzufügen und Entfernen von Füllzeichen erfolgt innerhalb der Ver-/Entschlüsselungs-Unterprogramme, so dass die Erstellung und Entfernung von Füllzeichen von Anwendungsprogrammen normalerweise nicht vorgenommen werden muss. COBOL-Anwendungsentwickler sollten sich jedoch über ungeeignete Methoden der Verwendung von Füllzeichen im Klaren sein.

Es gibt fünf Standard-Padding-Methoden, die heutzutage von Verschlüsselungssoftware verwendet werden. Die untenstehende Tabelle zeigt, wie der 12-Zeichen-String „HELLO WORLD!“ auf einen 16-Byte-Eingangsblock (in EBCDIC) aufgefüllt wird:

Methode 1: (PKCS#5, PKCS#7 & RFC3369)	Jedes Füllzeichen ist die Anzahl der hinzugefügten Füllzeichen (in hexadezimalen Format).	HELLO WORLD! ccddd4edddc50000 85336066934A4444
Methode 2:	Das am weitesten links stehende Füllzeichen ist hex „80“, und die folgenden Füllzeichen, sofern es sie gibt, sind jeweils hex „00“-Zeichen.	HELLO WORLD! ccddd4edddc58000 85336066934A0000
Methode 3:	Das am weitesten rechts stehende Füllzeichen ist die Anzahl der hinzugefügten Füllzeichen (in hexadezimalen Format), und die Füllzeichen davor, sofern es sie gibt, sind jeweils hex „00“-Zeichen.	HELLO WORLD! ccddd4edddc50000 85336066934A0004
Methode 4:	Jedes Füllzeichen ist hex „00“.	HELLO WORLD! ccddd4edddc50000 85336066934A0000
Methode 5:	Jedes Füllzeichen ist eine Leerstelle (hex „40“ in EBCDIC, hex „20“ in ASCII).	HELLO WORLD! ccddd4edddc54444 85336066934A0000

Die obige Tabelle zeigt, wie die Methoden 1 und 3 die genaue Anzahl der zu entfernenden Füllzeichen durch eine Untersuchung des Bytes an Position 16 sicher angeben. Zusätzliche Zeichen könnten jedoch bei den Methoden 2, 4 und 5 entfernt werden, wenn der Klartext auf Hex „80“, LOW-VALUES oder SPACES - in COBOL-Anwendungen übliche Werte - enden würde.

Hinweis: Das [Redvers Encryption Module](#)^[20] verwendet Methode 1 - [Public-Key Cryptography Standards \(PKCS#7\)](#)^[19] als Füllmethode für ECB-, CBC- und CFB-Modus-Verschlüsselung. Bei der Entschlüsselung wird jede der fünf Standardmethoden akzeptiert.

Anhang E: Exclusives Oder (XOR)

Der eXclusive Or (**XOR**)-Prozess vergleicht jedes der 8 Bits innerhalb eines Bytes aus einem String mit den entsprechenden 8 Bits innerhalb eines Bytes aus einem anderen String. Wenn beide Bits Null sind oder beide Bits eins sind, wird das Bit im Ergebnis auf null gesetzt. Andernfalls wird das Bit im Ergebnis Bit eins sein.

Beispiel:

```

Byte1:   1 0 0 1 1 0 1 1
Byte2:   0 1 0 1 0 0 1 0
-----
Ergebnis: 1 1 0 0 1 0 0 1
-----

```

Die Wirkung der **XOR**-Funktion ist, dass jedes Klartextzeichen direkt in ein Geheimtextzeichen umgewandelt werden kann, indem es mit einer Zeichenfolge von 8 Pseudozufallsbits mit XOR verknüpft wird. Wenn man diesen Vorgang für alle Zeichen einer Klartextzeichenfolge fortsetzt, erhält man einen vollständigen Geheimtext. Pseudozufallsbits können aus Münzwürfen, einem AES-Verschlüsselungsunterprogramm oder aus einer anderen sicher zufälligen Quelle erzeugt werden.

Die Entschlüsselung geschieht, indem der Geheimtext mit denselben Pseudozufallsbits mit XOR verknüpft wird, die für die Verschlüsselung verwendet wurden. Das Ergebnis ist der ursprüngliche Klartext.

Beispiel (unter Verwendung der Zeichen von oben):

```

Ergebnis: 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
-----

```

Das **XOR**-Verfahren ist recht einfach und sehr schnell, wenn es richtig optimiert wird. Redvers Consulting bietet eine kostenlose, optimierte, herunterladbare COBOL **XOR**-Unterroutine an: [RCNOTOR](https://www.cobol.de/data_encryption.php)^[24].

Anhang F: Referenzen

- [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.de/tools.php>
- [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.cobol.de/hashing_algorithm.php
- [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.cobol.de/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.de/tools.php>

Anhang G: Über Redvers Consulting

Redvers Consulting bietet seit 1988 erstklassige Produkte und Dienstleistungen für COBOL-Anwendungen. Unser Ansatz, die Software als Quellcode auszuliefern, ermöglicht unseren Kunden die Erfüllung ihrer geschäftlichen Anforderungen mit einer zuverlässigen, effizienten und perfekt integrierten Lösung.

Unsere Kunden sind überwiegend große Finanzdienstleister in Großbritannien und den USA. In zunehmendem Maße sind wir im deutschsprachigen Raum und auch in anderen Branchen tätig.

Da unsere Software als verschlüsselter Quellcode ausgeliefert wird, bieten wir Unterstützung für alle Hardwareplattformen und Betriebssysteme, für die ein COBOL-Compiler existiert - EBCDIC, ASCII, big endian und little endian.

Einige unserer Kunden:

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)
Landesbank Hessen Thüringen (DE)
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)

Kontakt: <https://www.cobol.de/contact.php>

Deutsches Büro:

Redvers Consulting Ltd
Scharfeneckweg 2,
50739 Köln,
Deutschland

Tel: +49 (0)221 1704 9000

Hauptbüro:

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

Tel: +44 (0)870 922 0633

Entwicklungsbüro:

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

Tel: +44 (0)208 522 7404