Blockchiffren Teil 3

Krypto II

VL 13

15.11.2018

Buch des Tages

Titel: Moderne Verfahren der Kryptographie

Autor(en): Beutelspacher, Schwenk, Wolfenstetter

Verlag: Springer (8. Auflage 2015)

Umfang: ca. 180 Seiten

Hinweise zum Inhalt:

Das Buch macht sich zur Aufgabe, eine besonders leicht verständliche Einführung in die Kryptographie zu liefern. Diesem Anspruch wird es abschnittsweise gerecht. Teilweise leidet die Verständlichkeit statt zu profitieren, wenn wichtige Details übersprungen werden. An anderen Stellen finden sich mathematisch anspruchsvolle Protokolle, deren Grundlagen nicht immer vorbereitet werden. Zur Abrundung einer kleinen Sammlung von Kryptographie-Einführungen ist dieses Buch dennoch geeignet. Als alleiniges Lehrbuch ist es nur mit Einschränkungen zu empfehlen.

Datenintegrität und Betriebsmodi

Bislang haben wir Betriebsmodi unter dem Gesichtspunkt betrachtet, ob und wie ein passiver Angreifer die Verschlüsselung knacken kann. Hierbei haben wir außer Acht gelassen, dass Schäden auch auf andere Weise entstehen können:

- unbeabsichtigte (Speicher- oder Übertragungsfehler) Veränderungen eines oder mehrerer Bits im Ciphertext
- beabsichtigte (aktive Angriffe) Veränderungen eines oder mehrerer Bits im Ciphertext
- beabsichtigte Auslassung oder Hinzufügung ganzer Ciphertext Blöcke
- beabsichtigte Veränderung der Reihenfolge von Ciphertext Blöcken

Daher schauen wir uns die einzelnen Modi an und bewerten, wie robust diese sind gegen Veränderung wie oben beschrieben. Bewertungskriterien sind dabei auch ***Error Propagation*** und ***Error Recovery***.

Data Integrity bei ECB

In ECB werden alle Blöcke unabhängig voneinander verschlüsselt. Insofern ist es möglich, Ciphertext Blöcke beliebig zu vertauschen, zu entfernen, oder wiederholt einzufügen. Inwieweit dies ggf. Sinn macht, ergibt sich aus etwaiger Kenntnis der Klartext-Struktur.

Verändert man ein (oder mehrere) Bits eines Ciphertext Blocks, so führt der Ver- bzw. Entschlüsselungsvorgang zur Diffusion über die gesamte Blockbreite und der entschlüsselte Plaintext Block ergibt nur Datensalat.

Betroffen ist jedoch stets nur der Block, in dem die Änderung stattfand.

Data Integrity bei CTR und OFB

In CTR werden alle Blöcke unabhängig voneinander verschlüsselt. Insofern ist es auch in CTR möglich, Ciphertext Blöcke beliebig zu vertauschen, zu entfernen, oder wiederholt einzufügen.

Verändert man ein (oder mehrere) Bits eines Ciphertext Blocks, so führt der Ver- bzw. Entschlüsselungsvorgang zur Diffusion über die gesamte Blockbreite und der entschlüsselte Plaintext Block ergibt nur Datensalat.

Betroffen ist jedoch stets nur der Block, in dem die Änderung stattfand.

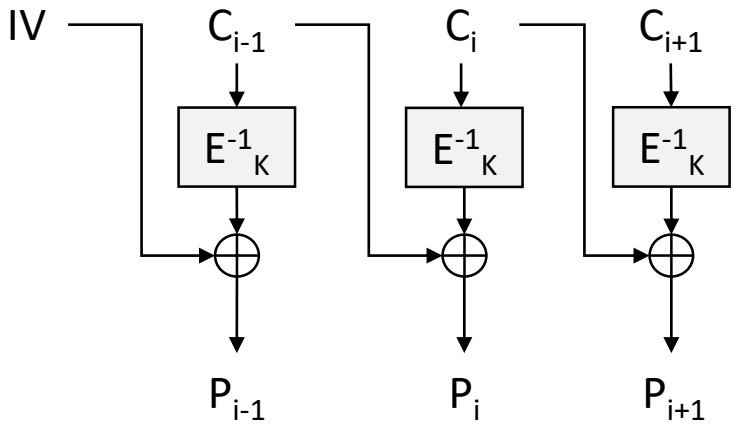
Kippt man ein Bit im Ciphertext block, so führt dies nach der Entschlüsselung zu einem gekippten Bit an derselben Position im Plaintext block.

Für unbeabsichtigte Änderungen ist es ein Vorteil, wenn sich die Auswirkungen auf die korrespondierenden Bitpositionen im Klartext beschränken.

Für beabsichtigte Manipulationen stellt dies jedoch ein Risiko dar, da gezielt einzelne Bits im Plaintext block verändert werden können.

Data Integrity bei CBC (1)

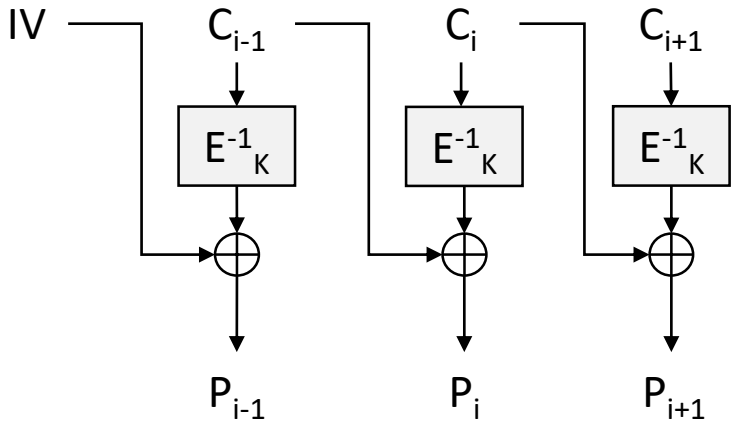
Cipher Block Chaining ist ein Modus, in dem die Veränderung eines oder mehrerer Ciphertext Bits offenkundig Auswirkungen auf benachbarte Blöcke hat. Schauen wir uns nochmals die Entschlüsselung an: ändert sich ein Bit in Ciphertext block C_i, so bewirkt die anschließende Entschlüsselungsfunktion eine Diffusion über den gesamten Output block und somit auch den Plaintext block P_i. Zusätzlich ändert sich in P_{i+1} das Bit an derselben Position. Auf weitere Plaintext blocks P_{i+2} etc. ergeben sich keine Auswirkungen. Entfernen, Tauschen oder Hinzufügen ganzer Blocks würde auffallen, da bei Entschlüsselung nur noch Datensalat entstehen würde. Wir könnten allerdings einzelne Bits in P_i kippen, indem wir die jeweiligen Bitpositionen in C_{i-1} kippen. P_{i-1} ergibt dann Datensalat, doch die Manipulation von P_i war erfolgreich.



Data Integrity bei CBC (2)

Ein radikalerer Eingriff ist folgendermaßen möglich: angenommen, ein Angreifer kennt Teile des Plaintexts. Dann könnte er einen Ciphertext C'_i (zu dem er den Plaintext kennt) einsetzen anstelle des ursprünglichen Ciphertexts C_i. Entschlüsselt der Empfänger nun den Ciphertext, so erhät er anstelle von P_i den ausgetauschten Plaintext P'_i.

Der Austausch von C_i durch C'_i führt allerdings dazu, dass der folgende Plaintext P_{i+1} nur Datensalat ergibt. Theoretisch könnte ein Angreifer so jeden zweiten Plaintext austauschen, dazwischen befindliche Plaintexts wären unleserlich.



Data Integrity bei CFB (1)

Ein Bitfehler in einem Ciphertext block führt bei CFB dazu, dass dieser Block und mehrere weitere fehlerhaft entschlüsselt werden. Die Fehler enden erst dann, wenn der fehlerhafte Abschnitt aus dem Register herausrotiert wurde.

Angenommen wir haben einen Bitfehler in Ciphertext block C_i. Es gilt:

$P_i = MSB_m(E_K(LSB_{n-m}(R_{i-1}) \| C_{i-1})) \oplus C_i$

In P_i ist daher auch nur an gleicher Position das Bit verändert. Der Rest bleibt OK.

Wie verhält es sich mit dem darauf folgenden Nachrichtenblock:

$P_{i+1} = MSB_m(E_K(LSB_{n-m}(R_i) \| C_i)) \oplus C_{i+1}$

Wie wir sehen, wird ein veränderter Inputvektor mittels Blockchiffre E_K verschlüsselt, der Output (also insbesondere auch dessen m Most significant bits) ist folglich über die gesamte Blockbreite verändert. Für P_{i+1} erhalten wir also nur Datensalat.

In jedem weiteren Entschlüsselungsschritt wird der defekte m-Bit Block C_i um m Positionen nach links geshiftet. Daher sind auch die nachfolgenden P_{i+2} etc. Datensalat, bis C_i komplett herausrotiert wurde.

Data Integrity bei CFB (2)

Beispiel: angenommen unsere Blockchiffre hat eine Blocklänge von n = 128 Bit und unser CFB Mode produziert Blöcke der Länge m = 8. Nehmen wir an, unser Ciphertext hat einen Bitfehler in Block C_i. Es gilt

$$P_i = MSB_m(E_K(R_i)) \oplus C_i = MSB_m(E_K(\underbrace{LSB_{n-m}(R_{i-1}) \| C_{i-1}}_{R_i})) \oplus C_i$$

Im Zustand R_i enthält das Register noch keinen Fehler.

Im Zustand R_{i+1} enthält das Register den fehlerhaften Block C_i rechts außen.

Unser m-Bit Block C_i nimmt insgesamt $\frac{n}{m} = \frac{128}{8} = 16$ verschiedene Positionen im Register an, bis er in Schritt 17 komplett herausgeshiftet wird.

In Schritt R_{i+17} erhalten wir also den ersten wieder fehlerfreien Inputblock für unsere Blockchiffre E_K. Register R_{i+1} bis R_{i+16} führen zu komplett falschen Plaintext blocks P_{i+1} bis P_{i+16}. Ab P_{i+17} verläuft die Entschlüsselung wieder fehlerfrei.

Verschlüsselung einzelner Datenfelder

Wir haben unterschiedliche Betriebsmodi kennengelernt. Bei der Anwendung denken wir bevorzugt an Nachrichten, die aus sehr vielen Blocks zusammengesetzt sind. Doch welches Verfahren sollen wir nehmen, wenn nur einer oder wenige Blöcke zu verschlüsseln sind?

Kommt in diesem Fall doch wieder ECB in Frage? Oder gibt es andere, speziell für “kurze” Nachrichten empfohlene Betriebsmodi?

Kann es zu Problemen führen, wenn ein kurzer Plaintext in einen längeren Ciphertext konvertiert wird, aufgrund des benötigten Initialisierungsvektors oder einer Nonce?

Finden sich geeignete Verfahren, die keinen Overhead benötigen in Form eines Initialisierungsvektors oder einer Nonce?

Wäre es eine Alternative, diverse Nutzdaten(felder) als eine längere Folge Blocks zusammenzuführen und dadurch nur einmal einen Initialisierungsvektor zu benötigen? Welche Betriebsmodi erlauben den dann erforderlichen Zugriff (Ver- und Entschlüsselung) auf beliebige Positionen in der Blockfolge?

Ciphertext Stealing (1)

Wir haben bereits kennengelernt, wie man mithilfe von Padding einen Plaintextstring “aufrunden”kann zur vollen Blocklänge. In der Folge ist der Ciphertext länger als der Plaintext.

Was können wir tun, wenn der Ciphertext nicht länger sein darf als der Plaintext? Ist dies überhaupt möglich? Wenn wir den letzten Plaintext block verschlüsseln, so müssen wir den gesamten hieraus resultierenden Ciphertext behalten. Anderenfalls ist die Entschlüsselung nicht mehr möglich. Zunächst klingt unsere Forderung daher nach einer unlösbaren Aufgabe.

Kryptographen sind jedoch immer für eine Überraschung gut, so auch hier: die Lösung lautet ***Ciphertext Stealing***.

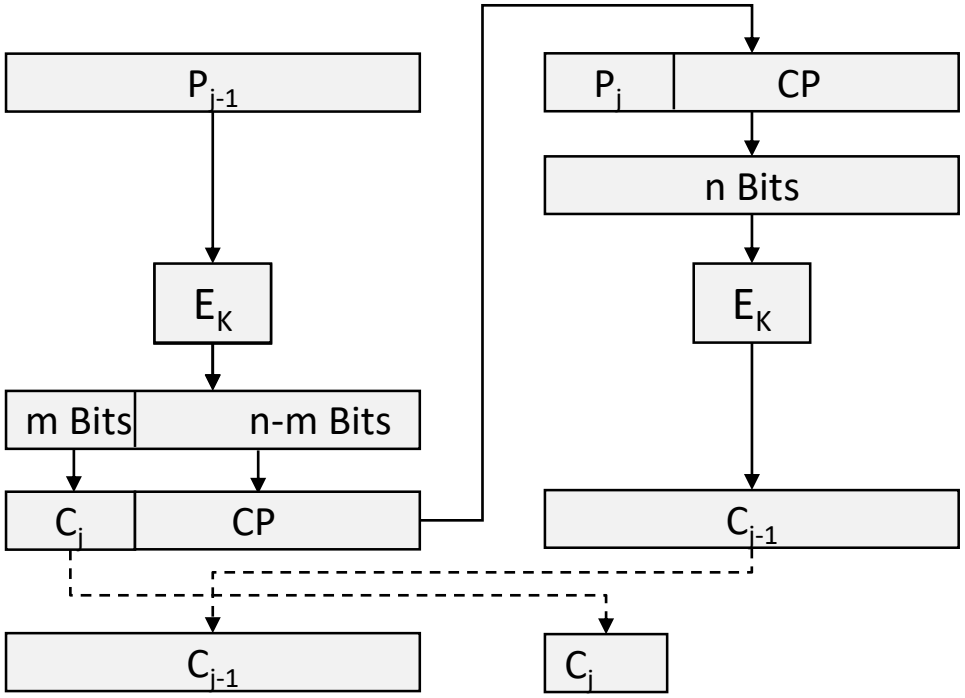
Angenommen die Blocklänge unserer Blockchiffre $E_K()$ sei n, und im letzten zu verschlüsselnden Plaintext block P_j seien nur noch m Bit Nutzdaten, $1 < m < n$.

Das Verfahren ist am einfachsten nachzuvollziehen anhand einer Graphik, die die beiden letzten Verschlüsselungsschritte für $E_K(P_{j-1})$ und $E_K(P_j)$ zeigt.

Ciphertext Stealing (2)

Den vorletzten Ciphertext block teilen wir auf in in einen Teilblock C_j der Länge m und einen Teilblock CP der Länge n-m. CP nutzen wir, um P_j aufzufüllen auf Blocklänge n. Nach Verschlüsselung erhalten wir wieder einen Block der Länge n. Diesen machen wir zu C_{j-1} , und unser C_j haben wir bereits aus dem vorherigen Verschlüsselungsschritt.

Bei der Entschlüsselung drehen wir die Prozedur einfach um.



Tweakable Block Cipher

Im Zusammenhang mit der Speicherung von Password Hashes haben wir eine Lösung kennengelernt, wie mithilfe eines sogenanntes Salt Wertes verhindert werden kann, dass zwei gleiche Passwörter auf denselben Output (Hashwert) abgebildet werden. Dies erschwert Angriffe, die Input-Output Paare in Tabellen speichern und später auswerten.

Ein ähnlicher Ansatz für Blockchiffren versucht zu verhindern, dass identische Plaintext Blöcke auf identische Ciphertext Blöcke abgebildet werden. Hierzu könnte man einfach den Cipher key der Blockchiffre ändern, doch dies ist eine kostenintensive (Zeit und Ressourcen) Operation. Deshalb führt man, analog zum o.g. Salt, einen zusätzlichen Parameter hinzu, ***Tweak*** genannt.

Dieser Tweak ist vergleichbar mit einem Initialisierungsvektor oder einer Nonce. Der Tweak ist allerdings nicht konzeptioneller Bestandteil des Betriebsmodus, sondern der zugrunde liegenden Blockchiffre. Diese nennt man dann auch ***Tweakable Block Cipher***. Deren Ver- und Entschlüsselungs-Routine benötigt drei statt zwei Input Parameter: den Cipher key, den Tweak, und den Plain- bzw. Ciphertext. Der Tweak muss nicht geheim gehalten werden. In der Praxis besteht er häufig aus einer laufenden Nummer, beispielsweise in Zusammenhang mit den Sektoren des jeweiligen Speichermediums.

Datenträgerverschlüsselung (1)

- Für die Verschlüsselung kompletter Datenträger hat man gesonderte Anforderungen im Vergleich zu anderen Anwendungen wie beispielsweise vertraulicher Datenübertragung oder der Verschlüsselung einzelner Dateien.
- Lese- und Schreibvorgänge müssen effizient sein. Manche Betriebsmodi kommen aus diesem Grund nicht in Frage. Ferner soll durch die Verschlüsselung nicht signifikant mehr Speicherplatz benötigt werden.
- Für die Sicherheitsbetrachtung nimmt man im allgemeinen an, dass potentielle Angreifer in der Lage sind
  - alle Sektoren des Datenträgers lesen zu können.
  - beliebige Daten vom System verschlüsseln und abspeichern zu lassen
  - ungenutzte Sektoren des Datenträgers zu beschreiben und sodann deren Entschlüsselung anfordern zu können.

Datenträgerverschlüsselung (2)

- Datenträger wie zum Beispiel Festplatten werden üblicherweise in sogenannte Sektoren aufgeteilt. Typische Größe für einen Sektor ist beispielsweise 512 Bytes bzw. 4096 Bit.
- Um den zuvor formulierten Sicherheitsanforderungen zu genügen, muss die Verschlüsselung “tweakable” sein (siehe Definition Tweakable Block Cipher):

  Angenommen ein Angreifer schafft es, einen verschlüsselten Sektor zu kopieren und diesen in einen anderen (zum Beispiel ungenutzten, oder ihm offiziell als User zugeordneten) Sektor zu schreiben. Dann könnte er die Entschlüsselung dieses Sektors veranlassen und erhält so den entschlüsselten Inhalt des kopierten Sektors.

  Deshalb lässt man die Position des Speicherortes bzw. Sektors in die Verschlüsselung mit eingehen, zusätzlich zum Encryption Key.

Datenträgerverschlüsselung (3)

- Die Verwendung einer Stromchiffre würde erzwingen, dass für jeden Sektor ein anderer Initialwert (“Schlüssel”) verwendet wird, um zu verhindern, dass derselbe Plaintext in unterschiedlichen Sektoren auf denselben Ciphertext abgebildet wird.
- Dies würde allerdings dazu führen, dass viel zusätzlicher Speicherplatz für die Ablage dieser Initialwerte benötigt würde. Ferner ist es ineffizient, ständig einen Schlüssel zu wechseln.
- Aus diesem Grund verwendet man zumeist nicht Stromchiffren sondern Blockchiffren bei der Datenträgerverschlüsselung.
- Da Sektoren größer sind als die Blocklänge einer Blockchiffre, ist man zusätzlich auf die Wahl eines geeigneten Betriebsmodus angewiesen.
- Die Verwendung von ECB-Mode scheidet (unter anderem) aus, weil dieser nicht tweakable ist.
- Die Verwendung von CTR-Mode scheidet aus, weil dieser die Eigenschaft einer Stromchiffre hat.

Datenträgerverschlüsselung (4)

- Die bisher betrachteten Betriebsmodi erlauben unterschiedlich gut, Vertauschungen oder Veränderungen an Blocks zu erkennen.
- Der erzielte Schutz bezieht sich allerdings in erster Linie auf “Data in transit”, also auf geschützt zu übertragende Daten. Ein Integritätsschutz für verschlüsselte Datenträger ist nochmals schwieriger.
- Alle bisher von uns betrachteten Betriebsmodi teilen das Problem, dass ein Einspielen eines früheren Zustands nicht automatisch erkannt würde. Spielt ein Angreifer beispielsweise ein älteres Backup ein um zu verhindern, dass eine vorgenommene Aktualisierung des Datenträgers Bestand hat, so fällt dieser Sachverhalt nicht auf.
- Um dies zu verhindern, wären andere Betriebsmodi und/oder Zusatzmaßnahmen erforderlich, beispielsweise mithilfe von Message Authntication Codes, Zeitstempeln und digitalen Signaturen.